論文 / 著書情報
Article / Book Information

題目(和文)	EHF帯における人体遮蔽に影響される電波伝搬チャネルの測定と予測技術の開発
Title(English)	Measurement and Simulation Techniques of Dynamic Radio Channel Property under Human Body Shadowing at EHF Band
著者(和文)	康哲嘉
Author(English)	Che Chia Kang
出典(和文)	学位:博士(工学), 学位授与機関:東京工業大学, 報告番号:甲第12889号, 授与年月日:2024年9月20日, 学位の種別:課程博士, 審査員:髙田 潤一,山下 幸彦,笠井 康子,秋田 大輔,青柳 貴洋
Citation(English)	Degree:Doctor (Engineering), Conferring organization: Tokyo Institute of Technology, Report number:甲第12889号, Conferred date:2024/9/20, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Measurement and Simulation Techniques of Dynamic Radio Channel Property under Human Body Shadowing at EHF Band

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Engineering

by

Chechia Kang

Supervisor

Professor Jun-ichi Takada

School of Environment and Society
Department of Transdisciplinary Science and Engineering
Global Engineering for Development, Environment and Society Course
Tokyo Institute of Technology

September, 2024

Acknowledgments

First of all, I would like to express my gratitude and thankfulness to my supervisor, Prof. Jun-ichi Takada, for his research guidance, suggestion, and support, and for providing me the opportunity to do great research and involve the research project in his laboratory. I also would like to thank Asst. Prof. Hang Song, Asst. Prof. Xin Du from Yamaguchi University, and Dr. Nopphon Keerativoranan for their kind discussion and tireless support in writing the journal paper. Without them, this work could not be finished. In addition, I would like to thank Prof. Kim from Niigata University for giving chance to involve in the research project and conduct the experiments in his laboratory. Likewise, I would like to thank researchers from European Cooperation in Science and Technology (COST) action CA20120, The Intelligence-Enabling Radio Communications for Seamless Inclusive Interactions (INTERACT) for their intended discussion and advice.

I would also like to thank the Japan-Taiwan Exchange Association for providing me the financial support during my master's and Ph.D. courses and the faculty and the student members of Tokyo Tech Academy of Super Smart Society for providing me with financial support and places for exercising my discussion skills. My achievement would not be possible without their fosterage.

Life as a researcher might have been tougher if I had failed to know these awesome people. I must express my deepest gratitude to my colleagues in the Lab as well as many friends with Dr. Panawit, Nishida, Murakami, Deepak, Jiang, and many more. Without their kindness, I could never adapt myself to Takada Lab. I also need to give thanks to my juniors for letting me train them as a senior. The experience of succeeding knowledge and skills are a must as a member of the academic society.

This thesis could not be complete without my beloved mom and dad, who sent me to Japan and always believed in my potential to finish this work. Last but not least, I appreciate all the kind people I have been involved in these ten years of my journey as a student.

Chechia Kang

June 2024

Abstract

The extremely high frequency (EHF) band enables the increasing demand for massive capacity of mobile communication in the super smart society. Since the communication link at the EHF band depends on the line-of-sight (LoS) channel, the LoS channel can be blocked by a moving human body occasionally. When the first Fresnel zone is fully blocked, the communication is cut off. To develop robust networking algorithms against human body shadowing (HBS), the HBS channel simulation and its experimental evidence are needed. There are simple human models with experimental evidence in conventional studies. Since the radius of the Fresnel zone is proportional to the wavelength, the detailed shape of the human body should be considered. Therefore, accurate human models and measurements need to be considered at the EHF band.

This thesis presents the synchronized measurement system, which consists of channel measurement and motion capture, and human models for HBS channel simulations. The proposed measurement system achieved an average synchronization error within 10 milliseconds, which is enough to validate the HBS simulation at the EHF band. Two types of human models were proposed, vertical long models and the model based on estimated diffraction paths. The vertical long models (i.e., the vertical screen models and an elliptic cylinder model) performed well in predicting the HBS duration and fading depth when the diffraction points only exist on the sides of the human body. The human model based on estimated diffraction paths was the extension to accommodate scenarios where the diffraction points could exist on the top of the human body, e.g., office and cabin. The proposals were evaluated by comparing the simulation result with the measurement result at 58 GHz and 300 GHz. As the contribution of this research, the accurate simulation for properties of the dynamic HBS channel with the experimental evidence serves as the foundation of the research of robust wireless communication systems at the EHF band against HBS.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

To satisfy the exponentially growing demand for mobile access with high throughput, the high-speed wireless communication system is one of the main development targets [11, 12, 13, 14]. The communication data rate of a propagation channel is related to the bandwidth availability and logarithmic value of the signal-to-noise ratio. Since the bandwidth availability is about one percent of the carrier frequency by the radio regulations [15], the key for the high-speed communication system is a high carrier frequency. Therefore, the frequency bands from 30 GHz to 300 GHz (extremely high frequency, EHF) have been used in 5G and the next generation of mobile communication systems. Radio waves in the EHF band feature high propagation loss. To compensate for the propagation loss, directive antennas are used in the wireless systems at the EHF band. Regarding the narrow pattern of the main beam of the array antennas, the wireless system at the EHF band relies on the line-of-sight (LoS) channel. However, the communication could be cut off when the LoS channel is shadowed. Considering the radio wave propagating from the transmitter (Tx) toward the receiver (Rx),

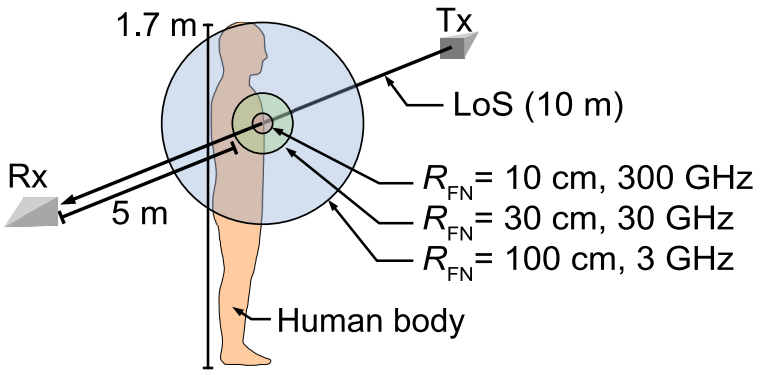

Figure 1.1: Comparison between the human body and the diameters of the Fresnel zones of various frequency bands.

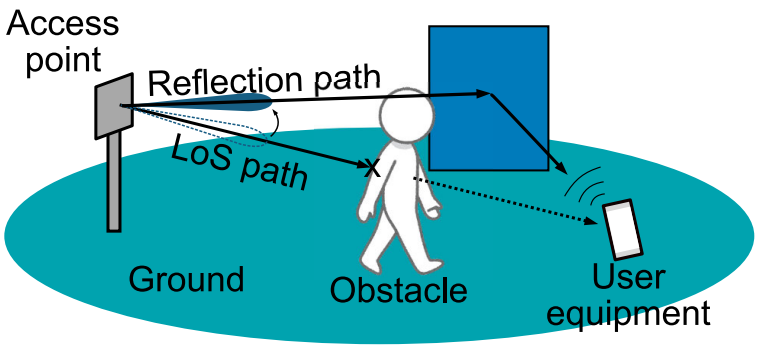

Figure 1.2: The LoS communication affected by the HBS. The networking algorithm points the main beam toward the reflection path to maintain the communication.

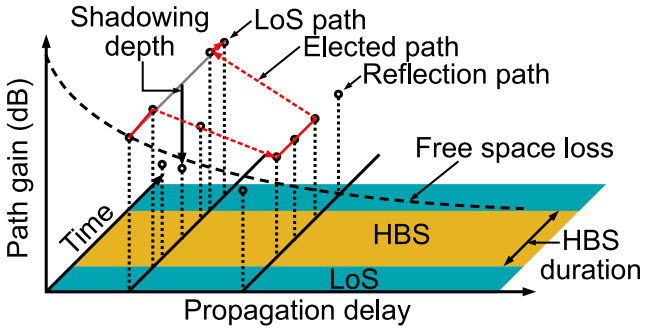

Figure 1.3: The ideal case of proactive scheduling. The access point switches the propagation path to the reflection path to keep the communication during the HBS event.

most of the power is condensed within the Fresnel zone, as shown in 1.1. The radius of the Fresnel zone is proportional to the square root value of the wavelength. For example, in an indoor scenario at 30 GHz, where the antenna separation is 10 m and the human obstacle is located 5 m from both antennas, the diameter of the first Fresnel zone R_{FN} is 30 cm. Since 30 cm is smaller than the average chest width of the human body [16], the indoor mobile communication system at the EHF band can be cut off while the LoS path is occasionally blocked by even a human body.

As shown in Fig. 1.2, the existence of reflectors contributing to alternative propagation paths is expected. Regarding the additional propagation length and reflection loss, the alternative paths have a larger propagation delay and more loss than the LoS path. Thus, as shown in Fig. 1.3, a robust networking system against human body shadowing (HBS) should switch the propagation path to the alternative one during the HBS event [17, 18].

For developing the networking system against HBS, a channel model simulating sufficient information for predicting the shadow event is needed. There are three types of channel models, empirical models, stochastic models, and deterministic models. The empirical models generalize the parameters of interest of a propagation channel by several parameters, [19]. The coefficients in the empirical models are defined based on plenty of channel mea-

surements. The stochastic models consider the parameters of a propagation channel as a random distribution when the propagation environment is rich in scattering waves. The deterministic channel models calculate the received radio wave by a combination consisting of the propagation theory and the corresponding geometry of the environment, [9]. With the experimental evidence, the deterministic models can be an alternative to the measurements for the empirical and stochastic models. Considering the shape of the human body changes due to various factors, e.g., age, gender, race, etc, the cost of the measurements is not tolerable. The stochastic models are not suitable for the HBS channel since the scattering waves from the human body interacting with the propagation path are too few to be treated as a random distribution. Thus, the scope of this work is to develop deterministic simulations for the HBS channel with experimental evidence. The result of this work serves for the further development of the empirical HBS channel models.

1.2 Research Motivation

In the deterministic simulation, a combination consisting of the propagation theory and the geometry is needed. For example, the most accurate simulation for the HBS channel model can be the set of the full-wave methods, e.g., the method of moments (MoM) [20], and the detailed human shape represented as a polygon [21]. Since the number of facets is proportional to the squared value of the frequency, the escalating computational complexity at the EHF band is not tolerable. Therefore, high-frequency asymptotic models using simplified human models are preferred. The high asymptotic methods should be validated by evidence based on full-wave simulation or measurement.

Since the full-wave simulations are not available due to the computational complexity, experimental evidence is mainly used [22, 19]. In the measurement, the dynamic HBS channel is measured by a channel sounder consisting of a set of the Tx and the Rx, which are set at where the access point (AP) and the user equipment (UE) are, respectively. The dynamic HBS channel is measured as the time-varying channel affected by the presence of the human body, as shown in Fig. 1.4. In the validation, the human model for the simulation should be matched with the human body in the measurement. To do so, a constant-speed walk or a motion capture (MoCap) system was used in literature. In the former, the human body in the measurement moved linearly with a fixed speed and with a fixed shape [22, 19]. Using the linear relation between the time and the location of the human body, the human model was matched to the human body by sliding the time offset between the measurement and the simulation. For example, a speed gun was used to help the specimen keep its speed when walking [19]. In [22, 23], a rail system was used to pull the specimen at a constant speed. However, the assumption is difficult to reach considering the short wavelength since the shape changes due to the breath and the gait is larger than the wavelength at the EHF

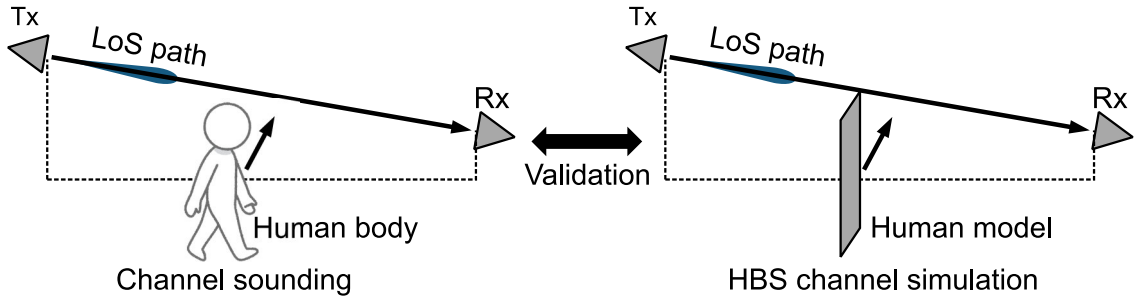

Figure 1.4: The validation for the simulation using a channel sounding. Comparing the results of the measurement and the simulation sharing the same propagation environment, the accuracy of the simulation could be validated.

band. Since the HBS channel properties, especially the shadowing duration, are sensitive to the instantaneous clearance of the first Fresnel's zone, the human model for simulation may lead to an inaccurate validation of the developed HBS channel models. On the other hand, MoCap systems based on light detection and ranging (LiDAR) were used to measure the human body. The depth camera in the MoCap system emits an infrared light (IR) signal and measures the time of flight (ToF) of the received signal in each pixel of the depth map. In [5, 8], the posture of the human body was measured by the attached IR markers on its joints. The human model was reconstructed by skinning the joints with screens or cylinders. Since the validations were conducted at the microwave band, the detailed difference between the human model and the human body was ignored, which is intolerable. In the simulation of the HBS channel, the clearance of the Fresnel zone should be recognized precisely. Thus, there is a challenge to match the surfaces of the human model and the human body. When the propagation environment is static, the point cloud (PC) representation was used to match the model and the measurement environment in [24, 25]. The reflection points on the walls and the ground were found by the PC and the normal vectors which calculated from the PC. Since the human body is dynamic, the PC-represented human body has to be synchronized to the HBS channel measurement, which is not developed.

As to the high-frequency asymptotic models for the HBS channel, source-based and ray-based asymptotic models using simplified human models can be considered. In the source-based models, the received field is calculated as the summation of the secondary sources from the human body. The secondary waves are considered from the facets of an arbitrarily shaped polygon in the physical optics (PO) [9], or are considered from the segments of the arbitrarily shaped edge in the modified edge representation equivalent edge currents (MER-EECs) [26, 27, 28]. The source-based approximations feature flexibility to the complex-shaped human body but need high computational cost and cannot address the propagation with multiple interactions with the interactive objects (IOs). In the ray-based models, the radio wave is treated as a ray and the obstacles are simplified as well-defined surfaces and

edges on the human models. The ray-based approximations feature low computational cost and allow the calculation of the propagation paths which experience multiple interactions with the IOs before arriving at the Rx. In the simplification of the human body, various fixed shapes have been used [4, 9, 29, 30, 31]. Since the human body with breath and gait is never a fixed shape, a low-cost HBS simulation with flexibility in the changing shape of the human body is needed.

By acknowledging these research gaps, this thesis aims to develop a measurement system for synchronized dynamic HBS channel and human motion and PC-based low-cost HBS channel simulation. The developed measurement system provides evidence for the proposed PC-based human models.

1.3 Objective and Contributions

The objectives of this study are to develop a synchronized measurement system consisting of the dynamic channel sounder and MoCap measurement and to develop PC-based human models for ray-based HBS channel simulation. To fulfill these goals, challenges described in the research motivation must be addressed in this thesis through the following novelties and contributions:-

- The novel synchronized measurement system consists of the channel sounder and the MoCap system that provides a PC-represented human body corresponding to dynamic channel response for validating the HBS channel model. (Chapter 3)
- Low-cost PC-based human model for simulating the shadowing duration and the shadowing gain in the shadow region that efficiently estimates the duration of the HBS event and the degradation of the channel during the HBS event. (Chapter 4)
- Robust PC-based human model for simulating the fading pattern in the lit region against various human motions that provides the Doppler frequency before/after the HBS event as a signal for the future networking algorithm. (Chapter 5)

1.4 Limitations of the Thesis

It should be noted that the PC-represented human body in the thesis is measured from the MoCap system consisting of the commercial-off-the-shelf (COTS) depth camera Microsoft Azure Kinect. Since the random error of the depth camera always exists, there is uncertainty in the generation of the PC-based models. Moreover, the framerate of a commercial depth camera is most limited at 30 frame-per-second (fps). Although Doppler frequency is proportional to the product of the carrier frequency and the speed of the human body, the

fading pattern could not be measured since the human specimen could not walk at extremely low speed. Another limitation is the undesired phase noise of the channel sounder. Since the high multiplying factors for generating the radio wave at the EHF band, the phase noise cannot be eliminated from the channel measurement. The remaining phase noise prevented us from conducting the short-time-Fourier-transform to obtain the Doppler frequency before/after the HBS event as a quantitative evaluation. As an alternative, we observed the prediction error in the shadowing gain. Furthermore, high-gain antennas were used to keep the dynamic range in the measurement at the EHF bands.

Regarding the HBS scenarios, this thesis considers the special case where the propagation channel is only shadowed by one person. Although a propagation channel may be blocked by multiple people in practice, the one-person shadowing assumption is an ideal and simple case to evaluate the accuracy of a certain human model. This assumption was well-used in literature to validate deterministic HBS models. Furthermore, the interference from the environment was not considered in the simulation since the high-gain antennas are expected in the communication system at the EHF band. At the frequency band lower than 30 GHz, the proposal may need modification for the multi-path components, which are also affected by the human presence.

1.5 Outline of the Thesis

The overall structure of this thesis is outlined as shown in Fig. 1.5. In this chapter, the motivation and background have already been provided.

Chapter 2 describes the propagation theories used in the ray-based simulation for the HBS channel. Then, the conventional measurements for the evidence of the HBS simulations are described. Finally, state-of-art human models for the HBS simulation are discussed in detail.

Chapter 3 firstly narrates the novel technique introduced to trigger the conventional channel sounder and the MoCap system simultaneously. Then, the structures of the channel sounders are described. Next, the MoCap system and the post-processing for generating the PC-represented human body are detailed. The performance was finally validated with the canonical problem, double knife-edges shadowing, where the LoS is shadowed by a vertical-long metal plate in the measurement. As a result, the synchronization accuracy is found enough to validate the HBS channel simulations.

Chapter 4 describes processes to generate the proposed low-cost PC-based vertical screen models of the human body for simulation. The proposals could naturally simulate the changing clearance of the Fresnel zone while the diffraction points exist on the sides of the human body. By using the Uniform theory of diffraction (UTD) method and the PC-based vertical screens, the dynamic HBS channel gain can be simulated. In the end, this chapter validates the PC-based vertical screen models by an indoor HBS channel measurement at

58 GHz. By comparing the simulation results of the shadowing duration and fading duration, it can be found the proposals are superior to the conventional fixed human model.

Chapter 5 describes processes to generate the proposed PC-based human body-shaped screen model of the human body for simulation. Considering the scenarios in the office and the cabin, the vertical screen model may be undefined, while the proposal shows flexibility in the motion of the human body. In addition to the UTD method, the source-based EECs method can be used with the PC-based human body-shaped screen to simulate the dynamic HBS channel gain. In the end, this chapter validates the PC-based vertical screen models by an indoor HBS channel measurement at 300 GHz. By comparing the simulation results, it can be found the proposal is superior to the vertical screen models in simulating the fading pattern in the lit region.

Finally, Chapter 6 concludes the thesis findings and provides prospective directions for future HBS channel models.

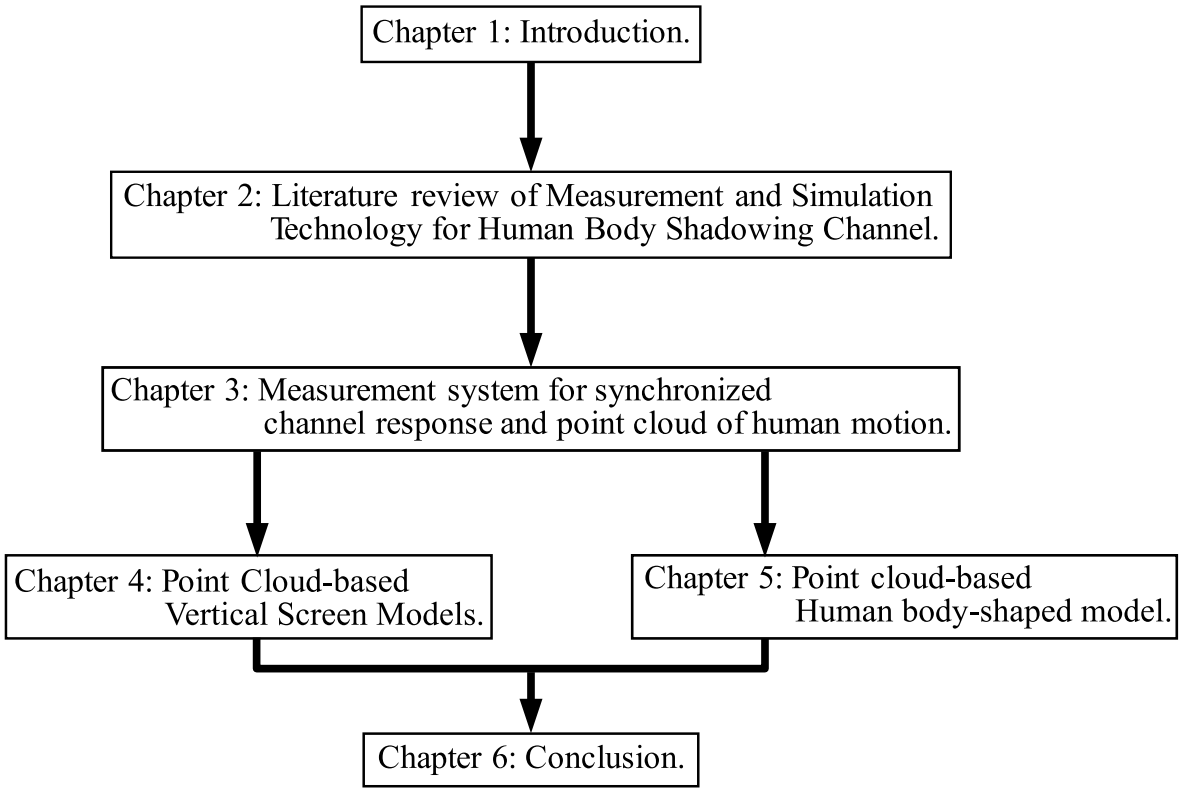

Figure 1.5: Structure of thesis

Chapter 2

Literature review of Measurement and Simulation Technology for Human Body Shadowing Channel.

2.1 Introduction

As the preparation for the shadowing gain simulation of the HBS channel, this chapter describes the propagation theories used in the simulation and the conventional works in the measurement and simulation techniques.

2.2 Propagation Mechenisms

The shadowing gain of the HBS channel is defined by the ratio of the dynamic channel gain affected by the HBS to the static channel gain of the LoS channel. The channel gain is the ratio of the received power to the transmitted power. Since the power of a radio wave is in direct ratio to the square of its electric field, the simulated shadowing gain $G^{\mathrm{s}}_{\mathrm{sim}}(t)$ in the propagation simulations is calculated as

$$G^{\mathrm{s}}_{\mathrm{sim}}(t) = 10\log_{10}\frac{|E^{\mathrm{Rx}}(t)|^2}{|E^{\mathrm{LoS}}|^2} \tag{2.1}$$

where $E^{\mathrm{Rx}}(t)$ and E^{LoS} are the instantaneous received field at time t and the direct field, respectively. The received field with the presence of the human obstacle is simulated by the propagation model and the suitable human model. In this section, the propagation mechanisms, the free space propagation, the UTD for edge diffraction, the UTD for absorbing edge, the UTD for the elliptic cylinder, and the MER-EECs methods are described.

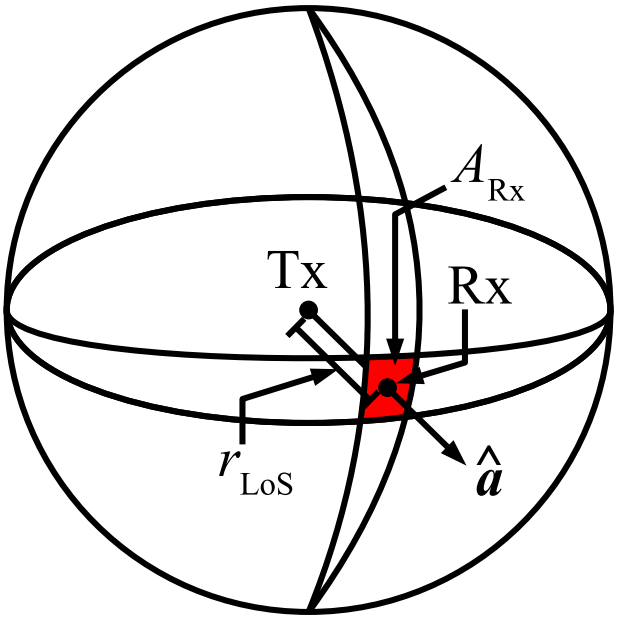

Figure 2.1: The free space propagation.

2.2.1 Propagation in Free Space

In the propagation environment where only the Tx and the Rx exist, the radio wave only experiences the free space loss. Considering isotropic antennas of the Tx and the Rx as shown in Fig. 2.1, the radio wave radiates toward all directions with a uniform power density. the free space channel gain is the ratio of the power density at the antennas of the Tx and the Rx without considering the antenna gain. Regarding the Huygens' principle, the actual source, which is the Tx antenna, can be replaced by an equivalent source on a closed surface. The received power P^{Rx} is the integral of the power density of the equivalent source of the Tx over the effective area of the Rx antenna, which is calculated as

$$P^{\mathrm{Rx}}(r_{\mathrm{LoS}}) = \int\int_{A^{\mathrm{Rx}}} \hat{\boldsymbol{r}} \frac{P^{\mathrm{Tx}} G^{\mathrm{Tx}}}{4\pi r_{\mathrm{LoS}}^2} \cdot \hat{\boldsymbol{a}} \mathrm{d}A \tag{2.2}$$

where $P^{\mathrm{Tx}}, G^{\mathrm{Tx}}, A^{\mathrm{Rx}}, r_{\mathrm{LoS}}, \hat{\boldsymbol{r}}$, and $\hat{\boldsymbol{a}}$ are the transmitted power, the of the Tx antenna, the aperture area of the Rx antenna, the antenna separation, the unit vector of the LoS direction, and the normal of the aperture surface of the Rx antenna, respectively. The integral of $\hat{\boldsymbol{a}}\mathrm{d}A$ in (2.2) is the effective area of the Rx antenna $A_{\mathrm{eff}}^{\mathrm{Rx}}$, which is calculated by [32]

$$A_{\mathrm{eff}}^{\mathrm{Rx}} = \frac{1}{4\pi\lambda^2} G^{\mathrm{Rx}} \tag{2.3}$$

where λ and G^{Rx} are the wavelength and the gain of the Rx antenna, respectively. The received power is then calculated as

$$P^{\mathrm{Rx}}(r_{\mathrm{LoS}}) = P^{\mathrm{Tx}} G^{\mathrm{Tx}} \left(\frac{\lambda}{4\pi r_{\mathrm{LoS}}} \right)^2 G^{\mathrm{Rx}}. \tag{2.4}$$

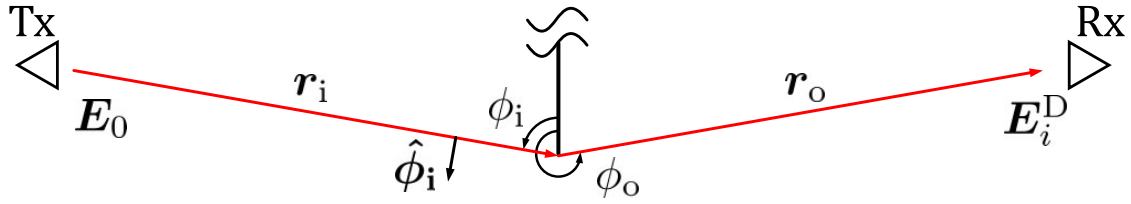

Figure 2.2: The top view of the diffraction from a vertical edge located between the TRx.

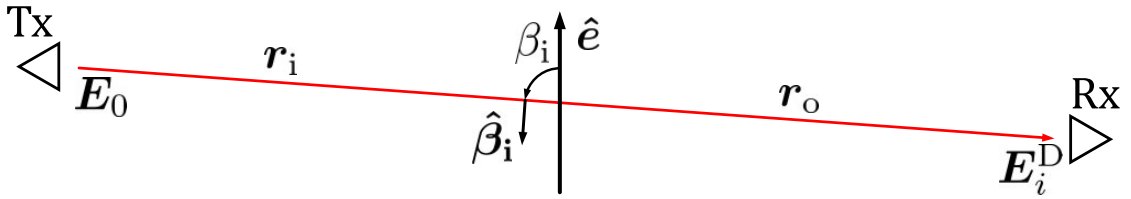

Figure 2.3: The side view of the diffraction from a vertical edge located between the TRx.

Since the amplitude of the electric field is proportional to the root value of the power of the radio wave, the relationship between the transmitted field $\boldsymbol{E}_\mathrm{Tx}$ and received field E_Rx considering the polarization can be written as

$$E_\mathrm{Rx} = \sqrt{G^\mathrm{Tx} G^\mathrm{Rx}} \boldsymbol{E}_\mathrm{LoS}(r_\mathrm{LoS}) \cdot \boldsymbol{u}_\mathrm{Rx}, \tag{2.5}$$

$$\boldsymbol{E}_\mathrm{LoS}(r_\mathrm{LoS}) = \boldsymbol{E}_\mathrm{Tx} \frac{\lambda}{4\pi r_\mathrm{LoS}} \exp(-\mathrm{j}k_0 r_\mathrm{LoS}) \tag{2.6}$$

where $\boldsymbol{E}_\mathrm{LoS}$ is the electric field induced by the Tx at the location of the Rx when both antennas are isotropic. The exponential term is the phase shift of the field along the direct path. k_0 and $\boldsymbol{u}_\mathrm{Rx}$ are the wave number in the free space and the polarization of the Rx antenna, respectively.

2.2.2 The Uniform Theory of Diffraction for Edge Diffraction

Considering an identified diffraction path from a knife edge, as shown in Figs. 2.2, 2.3. The parameters $\boldsymbol{u}_\mathrm{i,o}$ are unit vectors from the Tx to the edge and from the edge to the Rx, respectively. $s_\mathrm{i,o}$ are distances from the TRx to the edge, respectively. ϕ', ϕ are the angles of incidence and diffraction measured from the edge, respectively. $\hat{\boldsymbol{e}}$ is the unit vector of the edge, whose direction is counterclockwise from the viewpoint of the Tx. β_i is the incident angle of the oblique incident wave. $\hat{\boldsymbol{\beta}}_\mathrm{i}$ and $\hat{\boldsymbol{\phi}}_\mathrm{i}$ are the unit vectors of the parallel and perpendicular components of the incident fields, respectively. With the consideration of the polarization of the Rx antenna, $\boldsymbol{u}_\mathrm{Rx}$ the UTD method calculates the received field $E_\mathrm{UTD}^\mathrm{Rx}$ as the diffraction field from the edge and the direct field when the LoS is not blocked and as only the diffraction field when the LoS is blocked as

$$E_\mathrm{UTD}^\mathrm{Rx} = \begin{cases} \sqrt{G^\mathrm{Tx} G^\mathrm{Rx}} \boldsymbol{E}^\mathrm{LoS} \cdot \boldsymbol{u}_\mathrm{Rx} + \boldsymbol{E}_\mathrm{UTD}^\mathrm{D} \cdot \boldsymbol{u}_\mathrm{Rx}, & (\mathrm{LoS}) \\ \boldsymbol{E}_\mathrm{UTD}^\mathrm{D} \cdot \boldsymbol{u}_\mathrm{Rx}, & (\mathrm{NLoS}), \end{cases} \tag{2.7}$$

where $\boldsymbol{E}^{\mathrm{D}}_{\mathrm{UTD}}, \boldsymbol{E}^{\mathrm{LoS}}, G^{\mathrm{Tx,Rx}}$ and $G^{\mathrm{Tx,Rx}}_{\mathrm{UTD}}$ are the diffraction field from the edge, the direct field, the antenna gains of the boresight, and the directions toward the diffraction paths concerning the Tx and the Rx antennas, respectively.

The diffraction field $E^{\mathrm{D}}_{\mathrm{UTD}}$ is calculated as

$$\boldsymbol{E}^{\mathrm{D}}_{\mathrm{UTD}} = \sqrt{G^{\mathrm{Rx}}_{\hat{\boldsymbol{e}}}} \left(\hat{\boldsymbol{r}}_{\mathrm{o}} \times (\hat{\boldsymbol{r}}_{\mathrm{o}} \times \hat{\boldsymbol{e}}) \, E_{\beta_{\mathrm{o}}} + (\hat{\boldsymbol{r}}_{\mathrm{o}} \times \hat{\boldsymbol{e}}) \, E_{\phi_{\mathrm{o}}} \right), \tag{2.8}$$

$$E_{\beta_{\mathrm{o}}} = E_{\beta_{\mathrm{i}}} D \exp(-\mathrm{j}k_0 \left|\boldsymbol{r}_{\mathrm{o}}\right|) \sqrt{\frac{\left|\boldsymbol{r}_{\mathrm{i}}\right|}{\left|\boldsymbol{r}_{\mathrm{o}}\right| \left(\left|\boldsymbol{r}_{\mathrm{i}}\right| + \left|\boldsymbol{r}_{\mathrm{o}}\right|\right)}}, \tag{2.9}$$

$$E_{\phi_{\mathrm{o}}} = E_{\phi_{\mathrm{i}}} D \exp(-\mathrm{j}k_0 \left|\boldsymbol{r}_{\mathrm{o}}\right|) \sqrt{\frac{\left|\boldsymbol{r}_{\mathrm{i}}\right|}{\left|\boldsymbol{r}_{\mathrm{o}}\right| \left(\left|\boldsymbol{r}_{\mathrm{i}}\right| + \left|\boldsymbol{r}_{\mathrm{o}}\right|\right)}}, \tag{2.10}$$

$$E_{\beta_{\mathrm{i}}} = \hat{\boldsymbol{\beta}}_{\mathrm{i}} \cdot \boldsymbol{E}_{\mathrm{i}}, \tag{2.11}$$

$$E_{\phi_{\mathrm{i}}} = \hat{\boldsymbol{\phi}}_{\mathrm{i}} \cdot \boldsymbol{E}_{\mathrm{i}}, \tag{2.12}$$

$$\boldsymbol{E}_{\mathrm{i}} = \hat{\boldsymbol{r}}_{\mathrm{i}} \times (\boldsymbol{E}_0 \times \hat{\boldsymbol{r}}_{\mathrm{i}}) \sqrt{G^{\mathrm{Tx}}_{\hat{\boldsymbol{e}}}} H^{(2)}_0(k_0 \left|\boldsymbol{r}_{\mathrm{i}}\right|) \tag{2.13}$$

where $H^{(2)}_0(.)$ is the second kind of the Hankel function for the zeroth order. $\boldsymbol{E}_0$ and $\boldsymbol{E}_{\mathrm{i}}$ are the source field and the incident field at the edge, respectively. $E_{\beta_{\mathrm{o}},\phi_{\mathrm{o}}}$ and $E_{\beta_{\mathrm{i}},\phi_{\mathrm{i}}}$ are the parallel and perpendicular components of the diffraction and incident fields at the edge, respectively. The diffraction coefficients $D^{\mathrm{s,h}}$ depending on the polarization of the incident field on the edge in (25) of [33] is derived as

$$\begin{aligned} D^{\mathrm{s,h}}_{\mathrm{UTD}} = \; & \frac{-\exp\left(-\frac{\mathrm{j}\pi}{4}\right)}{4\sqrt{2\pi k_0}\sin\beta_{\mathrm{i}}} \\ & \times \Bigg[\cot\left(\frac{\pi+(\phi_{\mathrm{o}}-\phi_{\mathrm{i}})}{4}\right) \cdot F\left(k_0 L a^{+}\left(\phi_{\mathrm{o}}-\phi_{\mathrm{i}}\right)\right) \\ & + \cot\left(\frac{\pi-(\phi_{\mathrm{o}}-\phi_{\mathrm{i}})}{4}\right) \cdot F\left(k_0 L a^{-}\left(\phi_{\mathrm{o}}-\phi_{\mathrm{i}}\right)\right) \\ & \mp \cot\left(\frac{\pi+(\phi_{\mathrm{o}}+\phi_{\mathrm{i}})}{4}\right) \cdot F\left(k_0 L a^{+}\left(\phi_{\mathrm{o}}+\phi_{\mathrm{i}}\right)\right) \\ & \pm \cot\left(\frac{\pi-(\phi_{\mathrm{o}}+\phi_{\mathrm{i}})}{4}\right) \cdot F\left(k_0 L a^{-}\left(\phi_{\mathrm{o}}+\phi_{\mathrm{i}}\right)\right) \Bigg]. \end{aligned} \tag{2.14}$$

The function $F(\cdot)$ is the Fresnel integral calculated as

$$F(x) = 2\mathrm{j}\sqrt{x} \cdot \exp\left(\mathrm{j}x\right) \int_{\sqrt{x}}^{\infty} \exp\left(-\mathrm{j}t^2\right) dt. \tag{2.15}$$

where parameters L and $a^{\pm}$ are the calculated as

$$L = \frac{\left|\boldsymbol{r}_{\mathrm{i}}\right| \left|\boldsymbol{r}_{\mathrm{o}}\right|}{\left|\boldsymbol{r}_{\mathrm{i}}\right| + \left|\boldsymbol{r}_{\mathrm{o}}\right|} \cdot \sin^2 \beta_{\mathrm{i}}, \tag{2.16}$$

$$a^{\pm} = 2\cos^2\left(\frac{4\pi N^{\pm} - \beta}{2}\right), \tag{2.17}$$

$$\beta = \phi_{\mathrm{o}} \pm \phi_{\mathrm{i}} \tag{2.18}$$

where $N^{\pm}$ is calculated as the integer

$$N^{\pm} = \left[\frac{\beta \pm \pi}{4\pi} \right] \tag{2.19}$$

where function $[\cdot]$ is the rounding function to the nearest integer.

2.2.3 The Uniform Theory of Diffraction for Absorbing Half-plane

Since the Rx in the HBS scenario is never located in the reflection region, the reflection components, the latter two in the four terms in (2.14), can be mitigated [34]. Regarding the polarization making no difference in (2.14). The diffraction coefficient for an absorbing edge is simplified in [35] as

$$D^{\mathrm{s}}_{\mathrm{UTD}} = D^{\mathrm{h}}_{\mathrm{UTD}} = D_{\mathrm{AE}} = \frac{-\exp\left(\frac{-\mathrm{j}\pi}{4}\right)}{2\sqrt{2\pi k_0}\sin\beta_{\mathrm{i}}} \sec\left(\frac{\phi_{\mathrm{o}} - \phi_{\mathrm{i}}}{2}\right) F\left(2k_0 L\cos^2\left(\frac{\phi_{\mathrm{o}} - \phi_{\mathrm{i}}}{2}\right)\right). \tag{2.20}$$

2.2.4 The Uniform Theory of Diffraction for Elliptic Cylinder

Since the horizontal contour of the human body is similar to an elliptic cylinder, the UTD for a two-dimensional (2D) elliptic cylinder can simulate the shadowing gain of the HBS channel in scenarios where the heights of the TRx antennas are the same. In this case, the 2D simulation is used for the sake of simplicity, where

$$\beta_{\mathrm{i}} = \frac{\pi}{2}. \tag{2.21}$$

Consider a local coordinate system in the $u - v$ domain where the origin is the center of the ellipse and the major axis is the u axis, as shown in Fig. 2.4. When the LoS is blocked by the obstacle, the diffraction wave creeps through the surface of the ellipse. The incident point and the diffraction point $(r_1\cos\theta_{1,2}, r_2\cos\theta_{1,2})$ are found on the ellipse as

$$\theta_{1,2} = \arcsin\left(\frac{1}{\sqrt{\frac{u_{1,2}^2}{r_1} + \frac{v_{1,2}^2}{r_2}}}\right) - \arctan\left(\frac{u_{1,2} r_2}{v_{1,2} r_1}\right) \tag{2.22}$$

where (u_1, v_1) and (u_2, v_2) are the coordinates of the Tx and Rx. $\theta_{1,2}$ are the parameters of the incident point and the scattered diffraction point, respectively.

To predict the creeping diffraction coefficient when the Rx is in the shadow region, the geodesic from the incident diffraction point to the scattered diffraction point and the radii of the curvatures at both points are needed. The geodesic along an ellipse is the curve length $\widehat{r_m}$, which is derived as

$$\widehat{r_m} = r_2\left(\mathrm{E}(\theta_2|1 - \frac{r_1^2}{r_2^2}) - \mathrm{E}(\theta_1|1 - \frac{r_1^2}{r_2^2})\right) \tag{2.23}$$

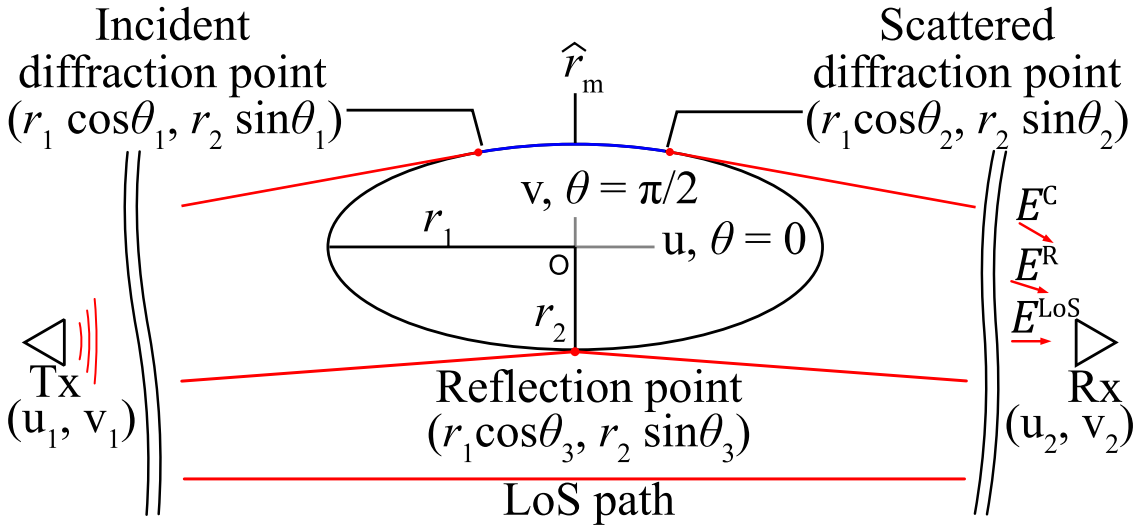

Figure 2.4: Diffraction from an ellipse located between the Tx and the Rx.

where E(·) is the incomplete elliptic integral of the second kind.The radii of the curvatures at the incident and scattered diffraction points, $\rho_{1,2}$, are calculated as

$$\rho_{1,2} = \frac{\sqrt{(r_1^4 r_2^2 \sin^2\theta_{1,2} + r_1^2 r_2^4 \cos^2\theta_{1,2})^3}}{r_1^4 r_2^4}. \tag{2.24}$$

The total received field from an ellipse $E^{\mathrm{Rx}}_{\mathrm{Elipse}}$ is calculated as the summation of the reflection and scattered diffraction fields from both sides of the ellipse and the direct field when the LoS is not blocked, and is calculated as only the summation of the scattered diffraction fields when the LoS is blocked as

$$E^{\mathrm{Rx}}_{\mathrm{Elipse}} = \begin{cases} \sqrt{G^{\mathrm{Tx}}G^{\mathrm{Rx}}}E^{\mathrm{LoS}} + \sqrt{G^{\mathrm{Tx}}_{\mathrm{C}}G^{\mathrm{Rx}}_{\mathrm{C}}}E^{\mathrm{C}} + \sqrt{G^{\mathrm{Tx}}_{\mathrm{R}}G^{\mathrm{Rx}}_{\mathrm{R}}}E^{\mathrm{R}}, & (\mathrm{LoS}) \\ \sum_{i=1}^{2}\sqrt{G^{\mathrm{Tx}}_i G^{\mathrm{Rx}}_i}E^{\mathrm{C}}_i, & (\mathrm{NLoS}) \end{cases} \tag{2.25}$$

where $E^{\mathrm{R}}, E^{\mathrm{C}}$, and $G_i^{\mathrm{Tx,Rx}}$ are the electric field of the reflection and diffraction paths from the ellipse, the antenna gains of the directions toward the diffraction paths concerning the Tx and the Rx antennas, respectively.

The creeping diffraction coefficient C can be considered as the absorbing-edge diffraction coefficient D in (2.20) added with an additional term mentioned in [36] as

$$C = D_{\mathrm{AE}} + \sqrt{M_1 M_2}e^{-\mathrm{j}k_0\widehat{r_m}}\sqrt{\frac{2}{k_0}}p^*(\xi^d, q^c_{s,h})e^{-\mathrm{j}\frac{\pi}{4}} \tag{2.26}$$

with

$$M_{1,2} = \left(\frac{k_0\rho_{1,2}}{2}\right)^{\frac{1}{3}}, \tag{2.27}$$

$$\xi^d = \left(\frac{k_0}{2}\right)^{\frac{1}{3}} \frac{(r_1 r_2)^{\frac{2}{3}}}{r_2} \left(\mathrm{K}(\theta_2|1-\frac{r_1^2}{r_2^2}) - \mathrm{K}(\theta_1|1-\frac{r_1^2}{r_2^2})\right), \tag{2.28}$$

$$q_{s,h}^c = -\mathrm{j}\sqrt{M_1 M_2}\left(\frac{\eta_0}{\eta}\right)^{\pm 1} \tag{2.29}$$

where $p^*(\cdot)$ is the associated Fock-type integral. $q_{s,h}^c$ are the parameters corresponding to the perpendicular (soft) polarization and the parallel (hard) polarization. η_0 and η are the free-space impedance and the surface impedance of the dielectric cylinder, respectively. $M_{1,2}$ and ξ^d are the UTD parameters mentioned in [37]. $\mathrm{K}(\cdot)$ is the incomplete elliptic integral of the first kind, respectively.

When the LoS is not blocked, the diffraction wave specularly reflects on the surface of the ellipse. The reflection point $(r_1\cos\theta_3, r_2\cos\theta_3)$, which satisfies the reflection law, is found on the ellipse numerically. θ_3 is the parameter of the reflection point.

To predict the reflection coefficient, the radius of the curvature at the reflection point is needed. The radius of the curvature at the reflection points, ρ_3, is calculated as

$$\rho_3 = \frac{\sqrt{(r_1^4 r_2^2 \sin^2\theta_3 + r_1^2 r_2^4 \cos^2\theta_3)^3}}{r_1^4 r_2^4}. \tag{2.30}$$

The reflection coefficient R can be considered as the edge diffraction coefficient D in (2.20) added with an additional term mentioned in [36] as

$$R = D_{\mathrm{AE}} + M_3\sqrt{\frac{2}{k_0}} p^*(\xi^r, q_{s,h}^r) e^{-\mathrm{j}\frac{(\xi^r)^3}{12}} e^{-\mathrm{j}\frac{\pi}{4}} \tag{2.31}$$

with

$$M_3 = \left(\frac{k_0\rho_3}{2}\right)^{\frac{1}{3}}, \tag{2.32}$$

$$\xi^r = -2M_3\cos\theta^{\mathrm{inc}}, \tag{2.33}$$

$$q_{s,h}^r = -\mathrm{j}M_3\left(\frac{\eta_0}{\eta}\right)^{\pm 1} \tag{2.34}$$

where $q_{s,h}^r$ are the parameters corresponding to the perpendicular (soft) polarization and the parallel (hard) polarization. M_3 and ξ^r are the UTD parameters mentioned in [37]. θ^{inc} is the incident angle.

2.2.5 Modified Edge Representation of Equivalent Edge Currents

As one of the source-based models, the EECs method flexible in complex-shaped obstacle is chosen as the reference since the computational complexity of the full-wave methods is

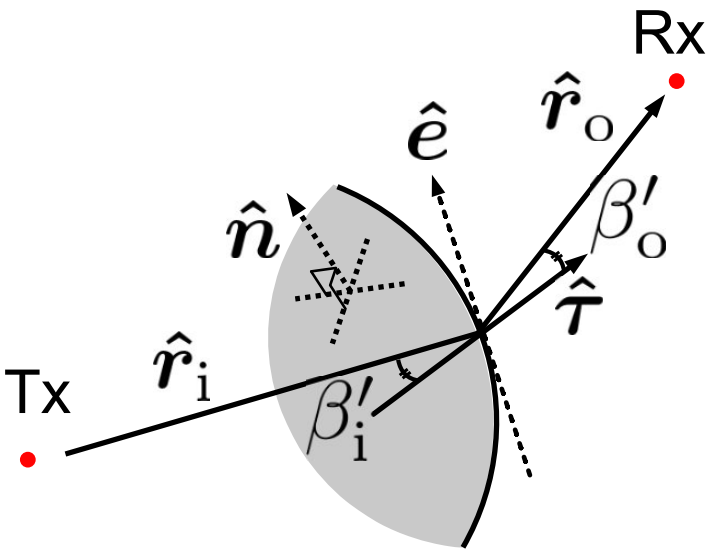

Figure 2.5: The modified edge representation $\hat{\boldsymbol{\tau}}$.

not tolerable at 300 GHz. The EECs method considers the total received field of the HBS channel as the summation of the fields from the induced EECs on the edges of an arbitrarily shaped screen. The EECs method is flexible in the arbitrarily shaped screen but leads to a higher computational complexity.

The EECs method calculates the received field $\boldsymbol{E}^{\mathrm{Rx}}_{\mathrm{EECs}}$ as the summation of the diffraction waves from the edge all around a plate as a line integral while the LoS is blocked, and the component of the direct wave is added while the LoS is not blocked as

$$\boldsymbol{E}^{\mathrm{Rx}}_{\mathrm{EECs}} = \begin{cases} \sqrt{G^{\mathrm{Tx}}G^{\mathrm{Rx}}}\boldsymbol{E}^{\mathrm{LoS}} + \boldsymbol{E}^{\mathrm{D}}_{\mathrm{EECs}}, & (\mathrm{LoS}) \\ \boldsymbol{E}^{\mathrm{D}}_{\mathrm{EECs}}, & (\mathrm{NLoS}) \end{cases}, \tag{2.35}$$

$$\boldsymbol{E}^{\mathrm{D}}_{\mathrm{EECs}} = \boldsymbol{E}^{\mathrm{D}}_{\mathrm{s}} + \boldsymbol{E}^{\mathrm{D}}_{\mathrm{h}} \tag{2.36}$$

where $\boldsymbol{E}^{\mathrm{D}}_{\mathrm{EECs}}$ and $\boldsymbol{E}^{\mathrm{D}}_{\mathrm{s,h}}$ are the summation of all the diffraction fields and the integral of the soft and hard polarizations, respectively. To calculate the diffraction fields from the edge all around the plate, the modified edge (ME) representation [26] is used. The modified edges are defined to satisfy the diffraction law for the given incident and diffraction directions as shown in Fig. 2.5. With a given combination of the unit vector of the incident and diffraction paths $\hat{\boldsymbol{r}}_{\mathrm{i}}$ and $\hat{\boldsymbol{r}}_{\mathrm{o}}$, the unit directional vector of the ME $\hat{\boldsymbol{\tau}}$ satisfies the conditions as

$$(\hat{\boldsymbol{r}}_{\mathrm{o}} - \hat{\boldsymbol{r}}_{\mathrm{i}}) \cdot \hat{\boldsymbol{\tau}} = 0, \tag{2.37}$$

$$\hat{\boldsymbol{n}} \cdot \hat{\boldsymbol{\tau}} = 0 \tag{2.38}$$

where $\beta'_{\mathrm{i,o}}$ are the incident and diffraction angle to the ME , respectively. $\hat{\boldsymbol{n}}$ is the normal vector of the illuminated face of the screen. $\hat{\boldsymbol{\tau}}$ is indefinite for the incident shadow boundary

(ISB) and reflection shadow boundary (RSB) conditions as

$$\hat{\boldsymbol{r}}_{\mathrm{o}} = \begin{cases} \hat{\boldsymbol{r}}_{\mathrm{i}}, & (\mathrm{ISB}) \\ \hat{\boldsymbol{r}}_{\mathrm{i}} - 2\hat{\boldsymbol{n}}(\hat{\boldsymbol{n}} \cdot \boldsymbol{r}_{\mathrm{i}}), & (\mathrm{RSB}) \end{cases}. \tag{2.39}$$

The ME in ISB and RSB conditions is defined as

$$\hat{\boldsymbol{\tau}} = \hat{\boldsymbol{e}}. \tag{2.40}$$

The sign of the $\hat{\boldsymbol{\tau}}$ satisfies the empirical rules in [27] as

$$\begin{array}{ll} \cos\frac{\phi'_{\mathrm{o}}-\phi'_{\mathrm{i}}}{2}\cos\frac{\phi_{\mathrm{o}}-\phi_{\mathrm{i}}}{2} > 0, & (\phi'_{\mathrm{o}} < \pi) \\ \cos\frac{\phi'_{\mathrm{o}}+\phi'_{\mathrm{i}}}{2}\cos\frac{\phi_{\mathrm{o}}+\phi_{\mathrm{i}}}{2} > 0, & (\phi'_{\mathrm{o}} > \pi) \end{array} \tag{2.41}$$

where $\phi'_{i,o}$ are the incident and diffraction angle for the ME, respectively. $\phi_{i,o}$ are the incident and diffraction angle for the actual edge $\hat{\boldsymbol{e}}$, respectively. The edge-fixed coordinate system for $\hat{\boldsymbol{\tau}}$ is shown as Fig. 2.6. $\hat{\boldsymbol{\beta}}_{\mathrm{i}}$ and $\hat{\boldsymbol{\phi}}_{\mathrm{i}}$ are the unit vectors parallel and perpendicular to the edge-fixed plan of the incident wave, respectively. The incident field $\boldsymbol{E}^{\mathrm{i}}$ at the point on the edge $\hat{\boldsymbol{\tau}}$ is decomposed into the components parallel to $\hat{\boldsymbol{\beta}}_{\mathrm{i}}$ and $\hat{\boldsymbol{\phi}}_{\mathrm{i}}$ as

$$\boldsymbol{E}^{\mathrm{i}}_{\beta_{\mathrm{i}}} = \hat{\boldsymbol{\beta}}_{\mathrm{i}} \cdot \sqrt{G^{\mathrm{Tx}}_{\hat{\boldsymbol{\tau}}}}\boldsymbol{E}_0 H_0^{(2)}(k_0\left|\boldsymbol{r}_{\mathrm{i}}\right|), \tag{2.42}$$

$$\boldsymbol{E}^{\mathrm{i}}_{\phi_{\mathrm{i}}} = \hat{\boldsymbol{\phi}}_{\mathrm{i}} \cdot \sqrt{G^{\mathrm{Tx}}_{\hat{\boldsymbol{\tau}}}}\boldsymbol{E}_0 H_0^{(2)}(k_0\left|\boldsymbol{r}_{\mathrm{i}}\right|) \tag{2.43}$$

where $G^{\mathrm{Tx}}_{\hat{\boldsymbol{\tau}}}$ and $\boldsymbol{r}_{\mathrm{i}}$ are the antenna gain of the Tx antenna in the direction toward $\hat{\boldsymbol{\tau}}$ and the vector from the Tx to $\hat{\boldsymbol{\tau}}$, respectively. The integrals $\boldsymbol{E}^{\mathrm{D}}_{\mathrm{s,h}}$ are calculated by integrating the diffraction wave from the tiny segments of the ME $\hat{\boldsymbol{\tau}}$ as

$$\boldsymbol{E}^{\mathrm{D}}_{\mathrm{s}} = \frac{\mathrm{j}\eta k_0}{4\pi}\oint \sqrt{G^{\mathrm{Rx}}_{\hat{\boldsymbol{\tau}}}}\hat{\boldsymbol{r}}_{\mathrm{o}} \times (\hat{\boldsymbol{r}}_{\mathrm{o}} \times \boldsymbol{I}_{\mathrm{s}})\frac{e^{-\mathrm{j}k_0|\boldsymbol{r}_{\mathrm{o}}|}}{|\boldsymbol{r}_{\mathrm{o}}|}\mathrm{d}l, \tag{2.44}$$

$$\boldsymbol{E}^{\mathrm{D}}_{\mathrm{h}} = \frac{\mathrm{j}\eta k_0}{4\pi}\oint \sqrt{G^{\mathrm{Rx}}_{\hat{\boldsymbol{\tau}}}}(\hat{\boldsymbol{r}}_{\mathrm{o}} \times \boldsymbol{I}_{\mathrm{h}})\frac{e^{-\mathrm{j}k_0|\boldsymbol{r}_{\mathrm{o}}|}}{|\boldsymbol{r}_{\mathrm{o}}|}\mathrm{d}l \tag{2.45}$$

$$\tag{2.46}$$

where $\boldsymbol{I}_{\mathrm{s,h}}$ are the EECs of the soft and hard polarization on $\hat{\boldsymbol{\tau}}$, respectively. $G^{\mathrm{Rx}}_{\hat{\boldsymbol{\tau}}}$ and $\boldsymbol{r}_i$ are the antenna gain of the Rx antenna in the direction toward $\hat{\boldsymbol{\tau}}$ and the vector from $\hat{\boldsymbol{\tau}}$ to the Rx antenna, respectively. The EECs $\boldsymbol{I}_{\mathrm{s,h}}$ on the edge at position $\boldsymbol{p}$ are calculated as

$$\boldsymbol{I}_{\mathrm{s}} = \frac{\mathrm{j}}{\eta k_0 \sin\beta_{\mathrm{i}}}\boldsymbol{E}^{\mathrm{i}}_{\beta_{\mathrm{i}}} D^{\mathrm{s}}_{\hat{\boldsymbol{\tau}}}\hat{\boldsymbol{e}}, \tag{2.47}$$

$$\boldsymbol{I}_{\mathrm{h}} = \frac{\mathrm{j}}{\eta k_0 \sin\beta_{\mathrm{i}}}\boldsymbol{E}^{\mathrm{i}}_{\phi_{\mathrm{i}}} D^{\mathrm{h}}_{\hat{\boldsymbol{\tau}}}\hat{\boldsymbol{e}}, \tag{2.48}$$

$$D^{\mathrm{s,h}}_{\hat{\boldsymbol{\tau}}} = \sec\frac{\phi'_{\mathrm{o}}-\phi'_{\mathrm{i}}}{2} \mp \sec\frac{\phi'_{\mathrm{o}}+\phi'_{\mathrm{i}}}{2} \tag{2.49}$$

where $D^{\mathrm{s,h}}_{\hat{\boldsymbol{\tau}}}$ are the diffraction coefficients based on the geometrical theory of diffraction (GTD), respectively.

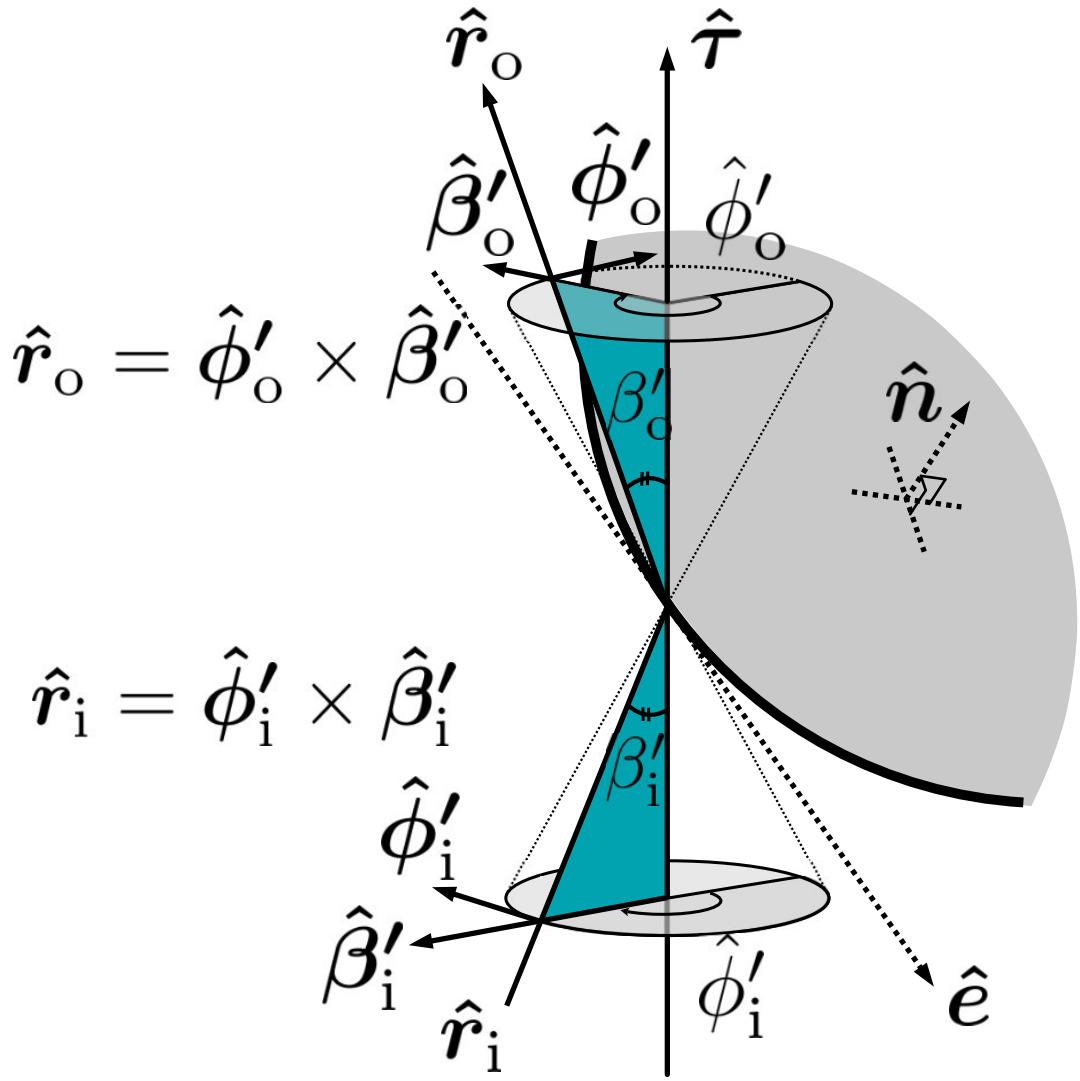

Figure 2.6: The coordinate system of the modified edge $\hat{\boldsymbol{\tau}}$.

2.3 Conventional Synchronized Measurement for Dynamic HBS Channel

As evidence for the empirical and deterministic models, dynamic channel sounding together with the measurement of the human body has been conducted. In the dynamic channel sounding for the HBS channel, the time-varying channel is measured in the time domain, which should be converted to the space domain for comparison with the human models. In this section, the methods for matching the domain, moving the specimen at a constant speed and synchronizing the channel sounder and the MoCap system are described.

2.3.1 Constant Speed Movement of the Specimen

Moving the specimen at a constant speed is a common method for converting the domain from time to space linearly in the measurement [38, 19, 22, 23]. Though this method benefits from simplicity, error exists whenever the motion of the human body has any difference from a rigorous straight trajectory with a steady speed and a fixed shape. To do so, the starting/stop point and the speed of the human body should be measured. In most cases, the starting/stop points are set by markers on the ground. In [38], the speed of the human body was obtained from the traveling time and the markers on the ground. In [19], a speed gun was used to help the specimen keep its speed when walking. In [22, 23], a rail system was used to pull the specimen at a constant speed. However, the assumption is difficult to reach considering the short wavelength at the EHF band since the shape changes due to the breath and the gait is near or larger than one wavelength, one centimeter. The centimeter accuracy in manually making the ground is not available, too. Since the HBS channel properties, especially the shadow duration, are sensitive to the instantaneous clearance of the first Fresnel's zone, the human model for simulation may lead to an inaccurate validation of the developed HBS channel models.

2.3.2 Synchronized Channel Sounder and MoCap

MoCap, such as conventional and depth cameras and the inertial measurement unit (IMU), have been used to record the motion of the human body during the measurement [5, 8, 39, 40, 41, 42]. By synchronizing the channel sounder and the measurement, the non-linear relationship between the time domain and the motion can be found. Regarding the rich degrees of freedom of human motion, such as the movements of different joints, the comparisons are conducted in the time domain or the space domain of a certain joint. In conventional research, the synchronization between the channel sounder and the MoCap was conducted by motion trigger or trigger signal. In [5, 8, 39, 40, 41], a motion trigger was used to synchronize the channel sounder for the body area network and the MoCap by a common sharp change of the channel and the motion. The common change is a motion, suddenly covering the Rx antenna with the hand, the hand gesture is recorded by the MoCap while a sharp fading is observed by the channel sounder. The motion trigger features versatility in that no specific hardware is used for synchronization. Therefore, the motion trigger can be applied to any combination of instruments. However, the accuracy of the synchronization was limited to the frame fate of the MoCap, which is too rough for the measurement at the EHF band. In [42], the synchronization was conducted by feeding the trigger signal from the MoCap, which is 30 fps, to a vector network analyzer (VNA) for the wideband channel sounding. Since every snapshot of the dynamic channel sounding was triggered by the MoCap, the synchronization accuracy is the resolution of the timer of the devices, which

is a microsecond. However, the sampling rate of the channel sounder is the same as the MoCap. Regarding the Doppler frequency of the short-term fading before/after the HBS event proportional to the carrier frequency, the sampling rate of the channel sounder should not be limited to the MoCap system.

2.4 Conventional HBS Models

With the high-frequency asymptotics, a human model is needed for the HBS channel simulation. In this section, various human models with fixed shapes in the literature, screen models, the hexagonal prism model, and the circular and elliptic cylinder models are described.

2.4.1 Screen Models

Screen models are the simplest shape for the human body. Since the shadowing channel is mainly defined by the clearance of the Fresnel zone, the screen models have been developed for efficiently simulating the shadowing gain of the HBS channel. There are various types of screen models as shown in Fig. 2.7. The vertical screen model is defined with a fixed width and infinitely long in the vertical direction [4, 5, 6]. In the simulation, the two paths from both sides are considered as the diffraction paths. Considering the thickness of the human body, the double vertical screen model is defined by two finite screens, which are perpendicular to each other [4, 5, 6]. Though there are four paths from the sides of the two screens, only the paths that are not shadowed by the other screen are considered in the simulation. To consider the diffraction wave from the top of the head, the top of the intersection contributes another diffraction path. In the cellular communication scenario, the difference in the width of the head and the shoulder is considered by the head-and-shoulder screen [6, 7]. In the simulation using the head-and-shoulder screen, four diffraction paths are considered, two from the side of the lower screen and the other two from the corner of the connecting part of the screens. Considering the HBS channel is dominated by the clearance of the Fresnel zone, the screen models are designed to efficiently calculate the shadowing gain using the cross-section of the viewpoint of the antennas. Among all the screen models, accurate prediction is achieved only when the direction of the incident wave is perpendicular to the screen. However, with fixed dimensions, the screen models achieve good agreement to the measurement only where the dimensions are measured. Considering the short wavelength, the discrepancy in the shape contributes to an error in simulating the shadow duration.

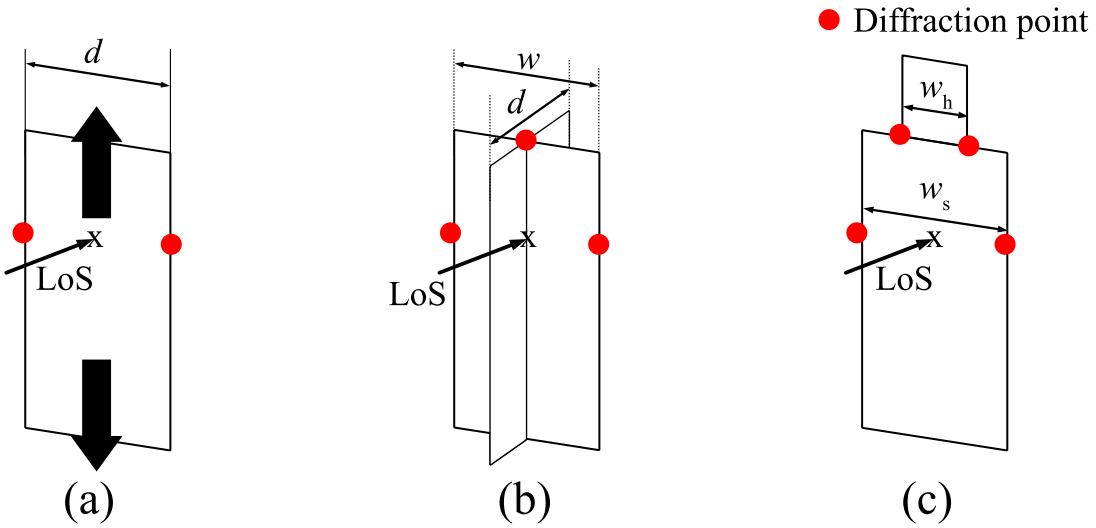

Figure 2.7: The screen type human models: (a) vertical screen [4, 5, 6], (b) double vertical screen [6], (c) head-and-shoulder screen [6, 7, 5].

2.4.2 Cylinder Models

There are various types of cylinder models as shown in Fig. 2.8. The circular screen model is defined with a fixed diameter [8, 9, 6]. In the simulation, the two paths travel from the Tx, creep along the surfaces of both sides and arrive at the Rx. The effect due to the skin is considered as the creeping loss when traveling along the surface. Considering the depth of the human body in the front direction is typically shorter than the width in the shoulder direction, a severe error in shadow duration is expected in the HBS simulation using the circular cylinder model. To address the ellipse-like horizontal contour of the human body, the elliptic cylinder is defined by the two axes [6]. However, it was difficult to validate the elliptic cylinder model since is difficult to match the directions of the human model and the human body in fine resolution. Another cylinder model dealing with the thickness of the human body is the hexagonal cylinder model. The hexagonal contour fits the shape of the human body better to achieve better accuracy than the double vertical screen model. Finally, the eleven-cylinder model was developed to achieve the best fit to the human motion in [8]. However, the accuracy was not improved even compared to the circular cylinder model. The cylinder-type models are designed to address the thickness of the human body. However, with fixed dimensions, the cylinder models also achieve good agreement to the measurement only where the dimensions are measured. Considering the short wavelength, the discrepancy in the shape contributes to an error in simulating the shadow duration. Moreover, the circular cylinder model caused an additional error in predicting the shadow duration.

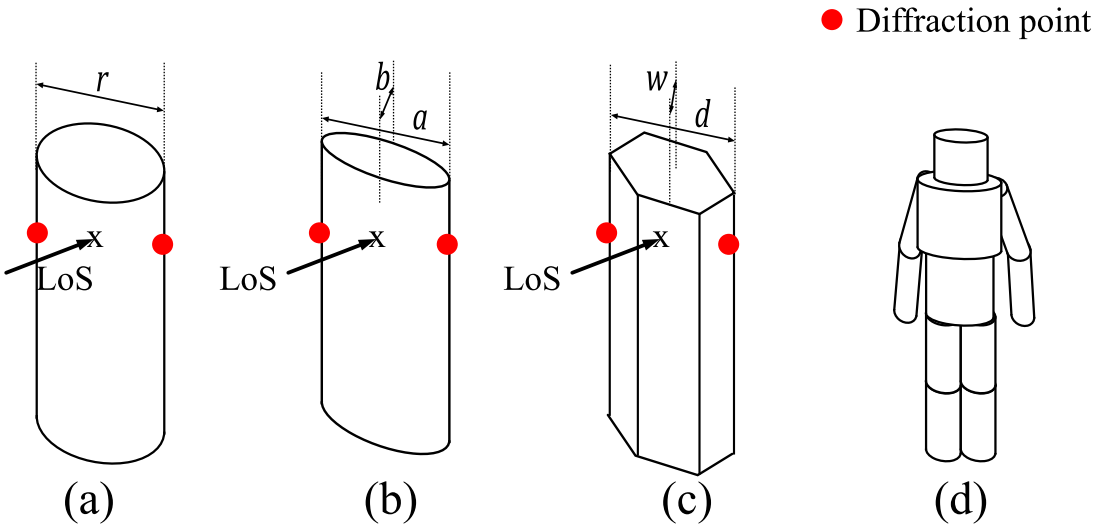

Figure 2.8: The cylinder type human models: (a) circular cylinder [8, 9, 6], (b) elliptic cylinder [6], (c) hexagon prism [9], (d) 11 cylinder model [8].

2.5 Summary

This chapter explained the mechanisms used in the HBS channel simulation, which was followed by the literature review of the synchronization technique for channel sounding and MoCap and the human models for HBS simulation. From the conventional measurement techniques, a research gap in synchronizing the channel sounder and MoCap with a high sampling rate in channel sounding exists. From the conventional human models for HBS channel simulation, it was found that current human models cannot accommodate the detailed shape of the human body. Regarding the short wavelength at the EHF bands, the prediction error especially in shadow duration is not tolerable.

Chapter 3

Measurement System for Synchronized Channel Response and Point Cloud of Human Motion

3.1 Introduction

As described in Section 2.3, a synchronized system consisting of a channel sounder and MoCap with a high sampling rate in channel sounding is needed for validating the developed HBS channel simulation. In this chapter, the proposed synchronized measurement system consisting of a channel sounder and MoCap system is described. Following the proposed synchronization approach, the structure and the signal processing are detailed. Finally, the experimental validation at 300 GHz is described.

3.2 Synchronization Approach [1]

To simultaneously obtain the channel gain and the shape of the human body, a shared trigger signal should be fed to both instruments, as shown in Fig. 3.1. The requirement of the instruments is a port for the trigger signal. The trigger port is typically installed in the Rx unit in the channel sounder, such as a signal analyzer, a digitizer, or a VNA. Since the camera-based MoCap systems are mainly designed for multi-camera configuration, the depth cameras are also equipped with a trigger port. By sending the trigger signal, both instruments start their measurements simultaneously. The trigger port of the Rx unit may be occupied for synchronizing the signal frame if the channel sounder was designed for wideband channel sounding [43]. If the Rx unit is a multi-channel digitizer, the reserved channel can be used for synchronization. In the measurement, the channel sounder starts the measurement at $t = 0$ before the MoCap system. When the trigger signal is sent, the

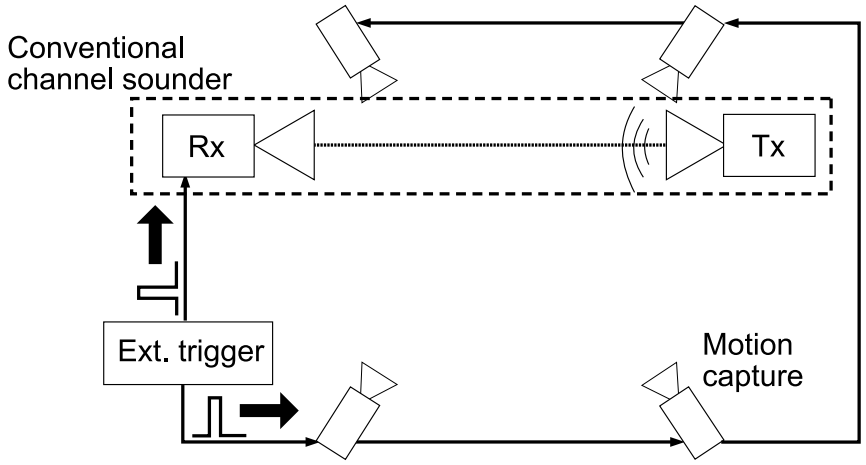

Figure 3.1: The hardware architecture of the synchronized channel sounder and the MoCap.

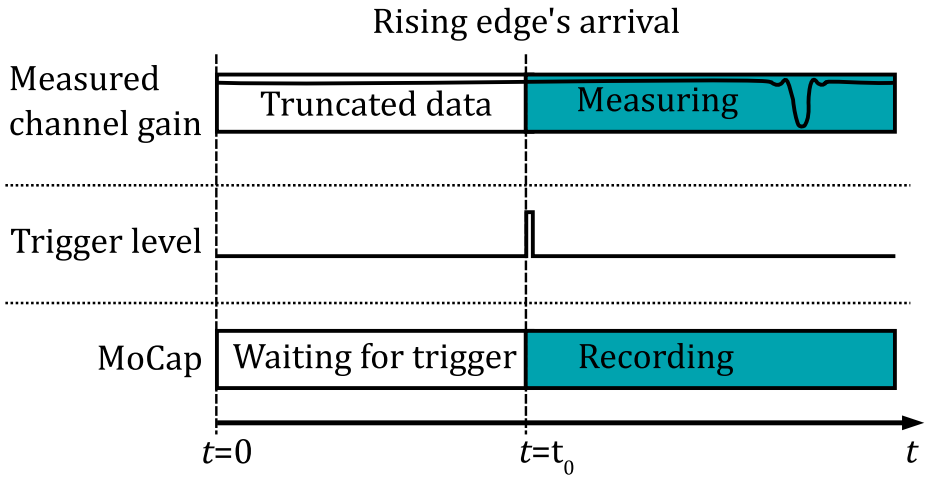

Figure 3.2: External rectangular pulse and the status of both instruments before and after receiving the trigger signal.

time of the sounder's clock t_0 is recorded. On the other hand, the MoCap system starts the recording of each camera immediately when the trigger is detected, as shown in Fig. 3.2. Therefore, the time offset from the MoCap system's clock to the sounder's is t_0.

3.3 Dynamic Channel Sounders

In the dynamic channel measurement, the shadowing gain $G^{\mathrm{s}}_{\mathrm{meas}}(t)$ is calculated as

$$G^{\mathrm{s}}_{\mathrm{meas}}(t) = 10 \log_{10} \frac{P^{\mathrm{Rx}}(t)}{P^{\mathrm{LoS}}} \tag{3.1}$$

where $P^{\mathrm{Rx}}(t)$ and P^{LoS} are the instantaneous received power at time t and the static received power without the human presence, respectively. To comply with the Radio Law in Japan,

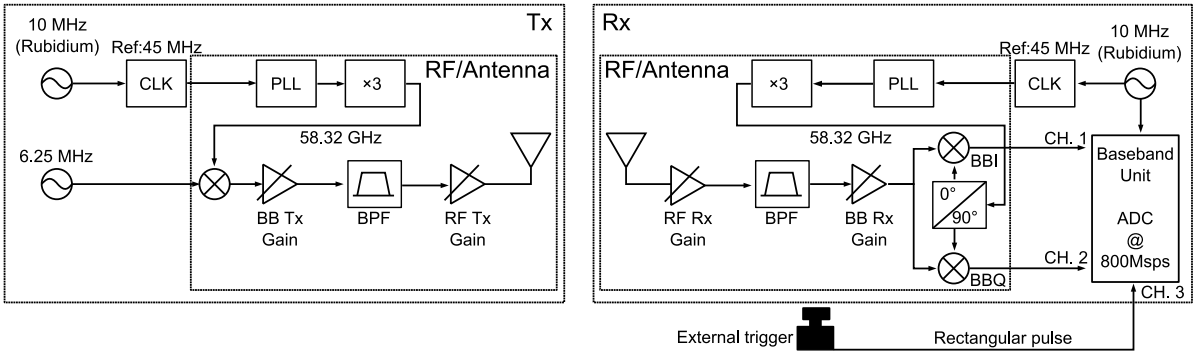

Figure 3.3: Hardware architecture of the channel sounder using COTS beamforming antenna at 58 GHz. The external trigger signal is fed to the third channel of the digitizer.

the dynamic channel sounders we could use in the EHF band were at the licensed 58 GHz and 300 GHz bands. Before we got the radio station license for operating the channel sounder at Tokyo Institute of Technology, we had to use the channel sounder at 58 GHz at Niigata University, Japan. In this section, the hardware architecture and the signal processing of the channel sounders are described.

3.3.1 Dynamic Channel Sounder at 58 GHz band

The channel sounder at 58 GHz is built by using a couple of COTS beam-forming antennas (Tx: BFM06005, Rx: RFM06010, Sivers IMA [44]) as the radio frequency (RF) front and a multi-channel baseband digitizer [43], as shown in Fig. 3.3. The Tx and Rx antennas are array antennas with 4 by 16 elements and 8 by 16 elements, which form narrow beams. The half-power beam widths (HPBW) of beam patterns are approximately 6° in the azimuth plane. The boresight transmission power is 41 dBm maximum in terms of equivalent isotropic radiated power (EIRP). The carrier wave at 58.32 GHz is heterodyne up-converted by a continuous wave (CW) at 6.25 MHz. After the heterodyne down conversion, the baseband I/Q signals were captured at the digitizer. What we refer to as a channel acquisition is the complex amplitude of the received I/Q signal. The upper tone was separated by using a fast Fourier transform. According to the description of the channel sounder [43], the dynamic range can be calculated from the signal-to-noise ratio (SNR) of the free space measurement and the dynamic range is 39 dB when the antenna separation is 7 m. For the normal walking speed $|v|$ is roughly 1 m/s, the maximum Doppler frequency $\nu_{\max}$ Hz is defined as

$$\nu_{\max} = \frac{|v|}{\lambda}\left(\hat{u}_{\mathrm{v}} \cdot \hat{u}_{\mathrm{Tx-p}} + \hat{u}_{\mathrm{v}} \cdot \hat{u}_{\mathrm{p-Rx}}\right) \tag{3.2}$$

where $\lambda, \hat{u}_{\mathrm{Tx-p}}, \hat{u}_{\mathrm{p-Rx}}$, and $\hat{u}_{\mathrm{v}}$ are the wavelength of the transmitted wave at 58.32 GHz, the unit vector point to the specimen from the Tx, the unit vector point to the Rx from the specimen, and the unit vector of the specimen's velocity, respectively. Regarding the narrow

Table 3.1: Specification of the channel sounder.

Carrier frequency	58.32 GHz
Signal bandwidth	12.5 MHz
EIRP	41 dBm
HPBW	Azimuth: 5.6° , Elevation: 45° (Tx), 18° (Rx)
Sounding signal	Unmodulated 2 tone
Sampling rate	40 MHz
FFT length	58752
Frame rate	680 fps

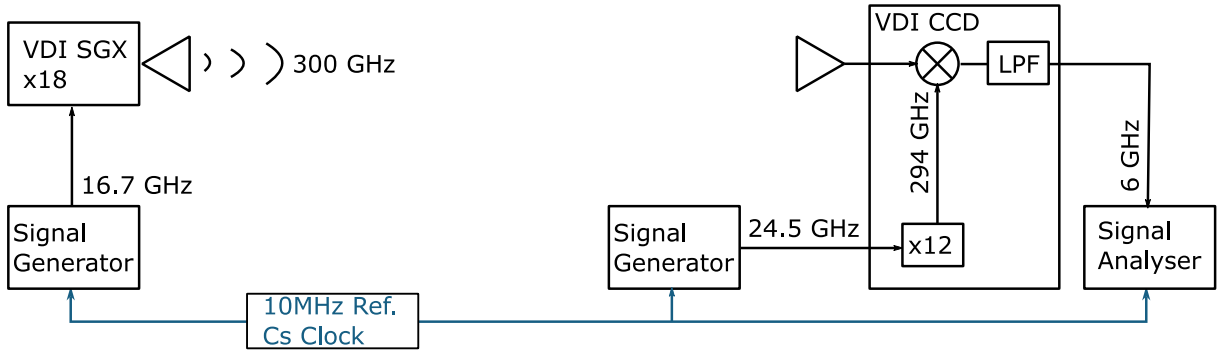

Figure 3.4: Hardware architecture of the channel sounder using COTS beamforming antenna at 300 GHz. The external trigger signal is fed to the trigger port of the signal analyzer.

beam width of the antennas, only the propagation paths within the first null positions of the beam pattern, $\phi_{\mathrm{Tx,Rx}}$, are expected to be received. When the specimen walks transverse to the LoS, the maximum of the Doppler frequencies can be estimated as

$$\nu_{\max} \leq \frac{2|v|}{\lambda} \cos\left(\frac{\pi - \phi_{\mathrm{Tx,Rx}}}{2}\right). \tag{3.3}$$

Since $\phi_{\mathrm{Tx,Rx}}$ are 20° [43], the receieved maximum Doppler frequency $\nu_{\max}$ is estimated not to exceed 68 Hz. To observe the HBS event including the complete fading pattern, the frame rate is set at 680 fps. The specifications of the channel sounder are given in Table 3.1.

3.3.2 Dynamic Channel Sounder at 300 GHz band

The channel sounder at 300 GHz is built by using a signal generator extension (VDI WR3.4SGX-M), a frequency down converter (VDI WR3.4CCD-M12) as the RF frontends, two signal generators as the local oscillators for the RF frontends (R&S SMF100, Keysight

Table 3.2: Specification of the channel sounder.

Carrier frequency	300 GHz
Sounding signal	CW
EIRP	16 dBm
HPBW	Azimuth: 9° , Elevation: 9°
Sampling rate	30 kHz
Resolution BW	24 kHz

E8267D), a signal analyzer to record the intermediate frequency signal, and a Caesium standard (Microsemi 5071A) as the reference signal for both TRx sides, as shown in Fig. 3.4. Horn antennas forming narrow beams to eliminate the multi-path components are used in TRx sites. Both antenna gains in the boresight direction are 26 dBi and the HPBW of the beam patterns are approximately 9° in both planes. The boresight transmission power is 16 dBm maximum in terms of EIRP. The Tx generates a 300 GHz CW signal from an up-converted CW at 16.7 GHz by frequency multiplier with a factor of 18. The received signal is down-converted to 6 GHz with the heterodyne converter and is observed by the signal analyzer. The local signal of the down converter is generated from a 24.5 GHz CW signal with a frequency multiplier with a factor of 12. All the signal generators and the signal analyzer share the 10 MHz reference signal generated from the Saesium standard. With high multiplying factors, the phase noise of the three RF devices cannot be removed. Therefore, only the power of the received signal is observed. The dynamic range found from the SNR of the free space measurement and the dynamic range is 54 dB when the antenna separation is 3.5 m. Calculated by (3.2), The maximum Doppler frequency $\nu_{\max}$ is estimated at 2 kHz while the human motion speed $|v|$ is at 1 m/s. To observe the HBS event including the Doppler frequency, the sampling rate of the signal analyzer is set at 30 kHz. The resolution bandwidth (RBW) is set at 24 kHz automatically, which is enough to observe the fading pattern due to human motion. The specifications of the channel sounder are given in Table 3.2.

3.4 Motion Capture System

To capture the surface of the human body corresponding to the dynamic propagation channel in the measurement, four synchronized depth cameras (Microsoft Azure Kinect) are used. The cameras are wired sequentially to share the trigger signal for synchronization. Based on LiDAR, the depth cameras measure the ToF of the IR signal, which is emitted by the

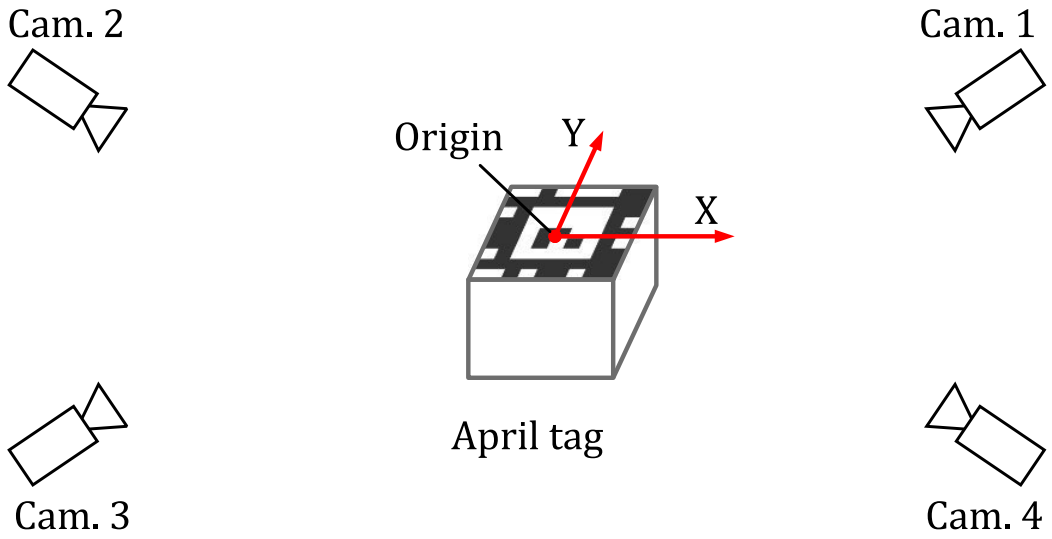

Figure 3.5: The April tag measurement. The tag is used to find the origin and the axes in each camera.

infrared flashlight and is scattered from the surface of the object. The depth d_{px} at each pixel is calculated by the ToF t_{ToF} as

$$d_{\mathrm{px}} = \frac{t_{\mathrm{ToF}}}{2c} \tag{3.4}$$

where c is the speed of the light. The depth maps are converted to the independent PCs measured by the viewpoints of four cameras. The independent PCs are then registered to the global coordinate system by the transforming matrices. Regarding the static configuration of the cameras, the transforming matrices are obtained by the MoCap measurement of a reference object. Since the MoCap system was under development during this work, various reference objects were used.

In the measurement at Niigata University, An April tag [45, 46] was used as the reference object. The April tag is placed at the center of the experiment field, as shown in Fig. 3.5. The position and direction of the April tag are used as the reference to find the transformation matrix between the two depth cameras' local coordinate systems. Using the transform matrices, the four PCs are transferred to the complete PC on the surface of the specimen as shown in Fig. 3.6. To make the April tag visible to all the cameras, the tag should be placed horizontally. However, it is difficult to prevent the reflection of the lighting or sunlight from existing on the tag. Since the reflection of the lighting in the camera cannot be mitigated, the derivation of the calibration matrices frequently failed, as shown in Fig. 3.7.

To avoid the issue of the April tag, laser spots were used as a new reference. Defined by transform and rotation, a calibration matrix $M_i(x, y, z, \theta_{\mathrm{x}}, \theta_{\mathrm{y}}, \theta_{\mathrm{z}})$ has 6 degrees of freedom. On the other hand, the degrees of freedom f_n of n ordered points is given as

$$f_n = 3n - 1. \tag{3.5}$$

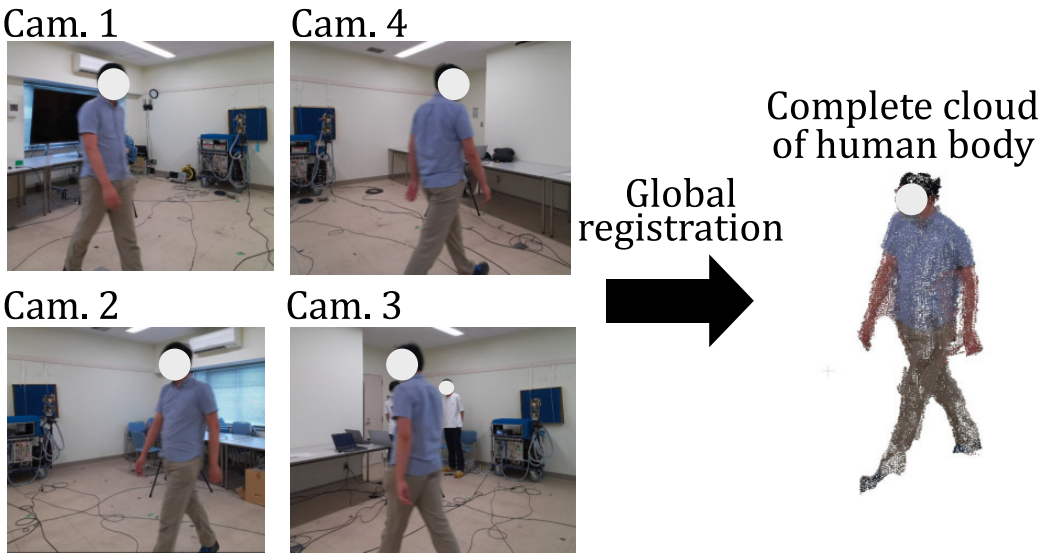

Figure 3.6: Example of the global registration for the four captured PCs.

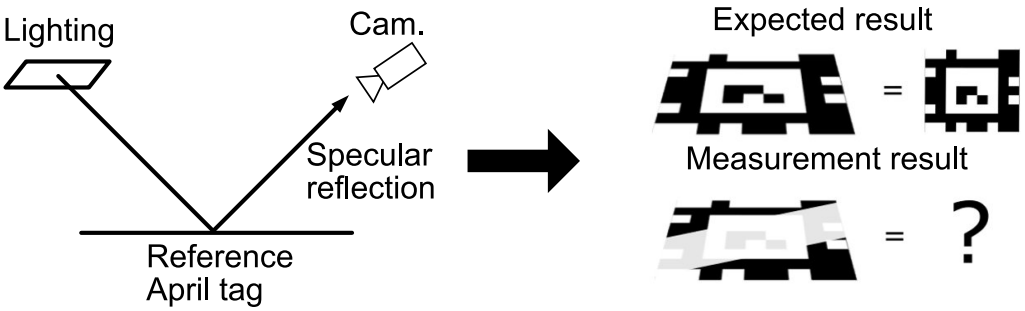

Figure 3.7: Example of the reflection of the lighting on the April tag.

That is, with 3 common points in the local coordinate system of each camera, the calibration matrices can be derived. Laser spots are used as the 3 common ordered points shared by the depth cameras, as shown in Fig. 3.8. The laser spot is generated by projecting the laser beam on thin pigment to make the laser spot visible to the cameras, as shown in Fig. 3.9. The laser spot method is used for the measurement validating the synchronized measurement system.

However, the laser spot method is also fragile to the reflection of the lighting and sunlight. Finally, the calibration square with IR markers is used [47]. As shown in Fig 3.10, with the normal vector pointing to the depth camera, the surface closest to the depth camera of each IR reflector is observed as the brightest pixel. Therefore, the position of each IR reflector is estimated as the position of the brightest IR pixel in the depth map added with the radius of the reflector. The calibration matrices are extracted by fitting the three estimated centroids of the reflectors to the dimension of the calibration square.

Regarding the random ranging error of the LiDAR measurement, the surface of the human body can be misestimated outward. With the standard deviation of the ranging error, which

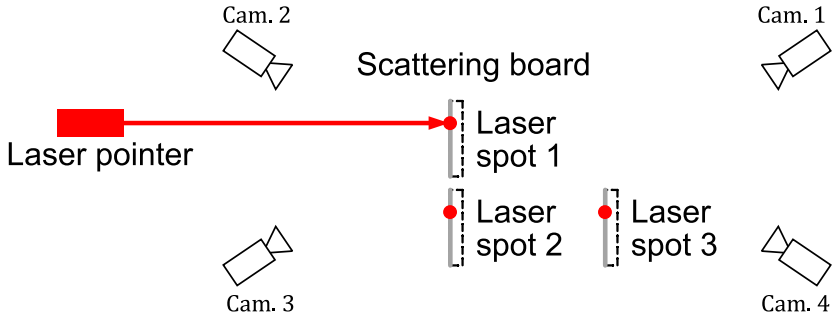

Figure 3.8: The MoCap measurement of the ordered laser spots.

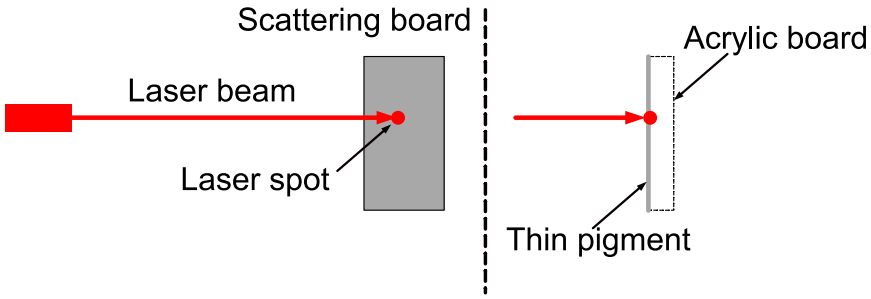

Figure 3.9: Laser spot generated by projecting the laser beam on the thin pigment.

is less than 17 mm [48], the Pr-th percentile c_{Pr} in the probability function $\mathrm{P}(\cdot)$ of the measured range can be represented as

$$\mathrm{P}(X < c_{Pr}) = Pr \tag{3.6}$$

where X is the measured value of the dimension of the PC-based model. By assuming that the measured range of the surface distributes Gaussian, the 97.5th percentile of the distribution of the measured range that $Pr = 97.5\%$ can be expressed according to the empirical rule in statistics as

$$c_{Pr} = \mu + \sigma_{\text{model}} \tag{3.7}$$

where μ and σ_{model} are the population mean and the standard deviation of the dimension of the PC-based model, respectively. Considering there is an angle θ_{M} between the boresight of the depth camera and the line of sight of the TRx as shown in Fig. 3.11, σ_{model} can be related to the standard deviation of the ranging error σ_{range} as

$$\sigma_{\text{model}} = \sigma_{\text{range}} \sin(\theta_M). \tag{3.8}$$

Considering the depth cameras were placed in the four corners, it can be found that $\theta_{\text{M}} = \frac{\pi}{4}$. Finally, with a 97.5% confidence level, the outward misestimation $c_{97.5}$ of the dimensions of the PC-based models is less than 24 mm.

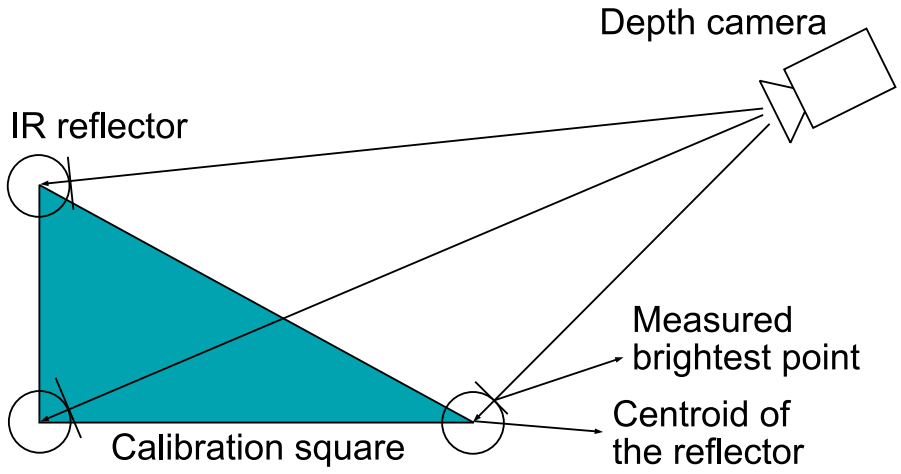

Figure 3.10: Calibration square based global registration.

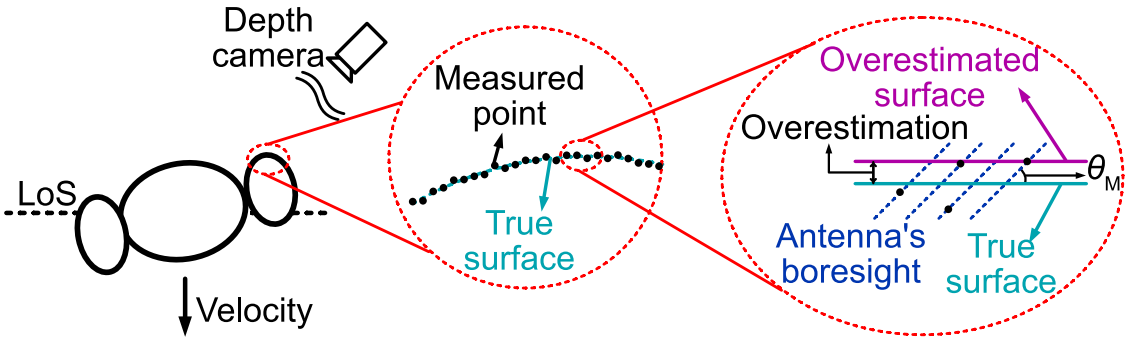

Figure 3.11: The outward misestimation of the position of the human surface. Since the contour points are the outermost points among their neighbors, the outward misestimation is the maximum of the ranging error toward the depth antenna.

3.5 Validation

To validate the synchronization accuracy of the proposed system, a measurement of a canonical problem, knife-edge diffraction, at 300 GHz was conducted. In this section, we compared the time in UTD simulation and measurement when the shadowing gain decays/rises across the threshold when the LoS was shadowed by a vertically long and wide metal plate. Since the rigorous solution of the shadowing gain when a half-plane just blocks half of the channel is -6 dB, the value of the threshold was set as -6 dB.

3.5.1 Measurement Environment

A synchronized dynamic HBS channel measurement and MoCap in an indoor scenario were considered. The measurement was conducted in a meeting room of Tokyo Institute of Technology. The set-up diagram of the HBS channel measurement in the office environment is

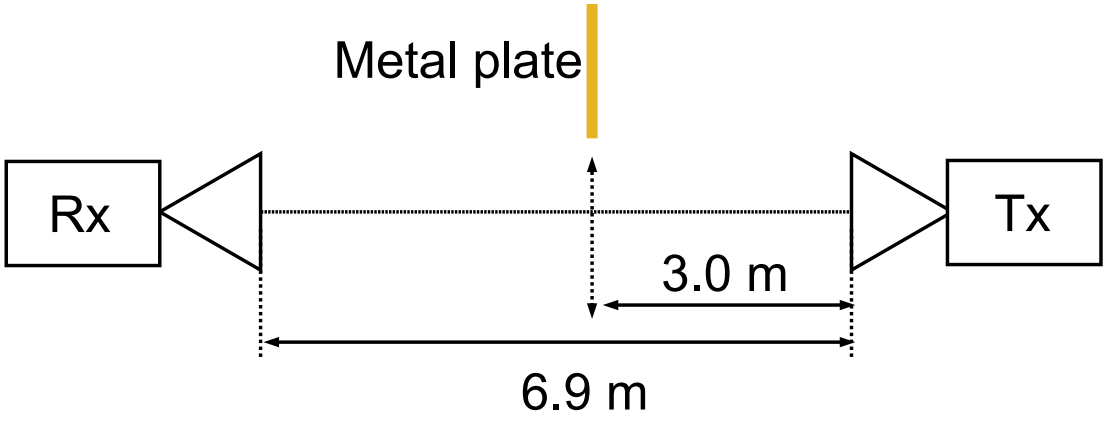

Figure 3.12: Diagram of measurement set-up with a metal plate traveled across the LoS for 1.5 rounds.

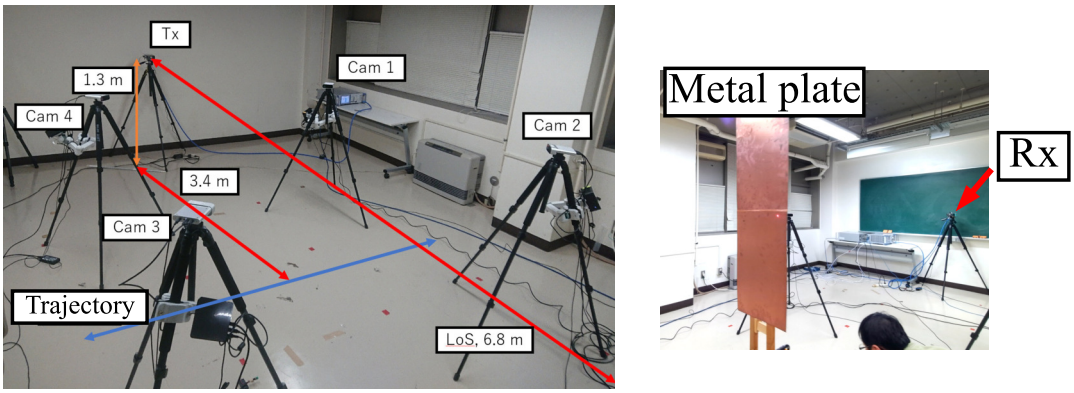

Figure 3.13: Photographs of the measurement.

shown in Fig. 3.14. The photographs of the environment are shown in Fig. 3.13 The multi-path components (MPCs) from the environment were excluded by using the narrow beam antennas. The Tx and the Rx were set at the same height for the sake of simplicity, where the TRx heights were 1.3 m. The antenna separation was 6.9 m. The metal plate was 400 mm wide and 1500 mm long. Since the first Fresnel radius at the intersection of the LoS and the route was 45 mm, it can be considered when the LoS interacted with the edge of the plate, the diffraction wave from the other edge was weak enough and could be ignored. The metal plate traveled 1.5 rounds across the LoS at the center of the LoS. In total, the measured shadowing was expected to decay/rise across the threshold -6 dB six times.

3.5.2 Results and Discussion

The results of the measurement and the UTD simulation are obtained in Tab. 3.3. The shadowing gain of one of the shadow events is shown in Fig. 3.14. The mean absolute error

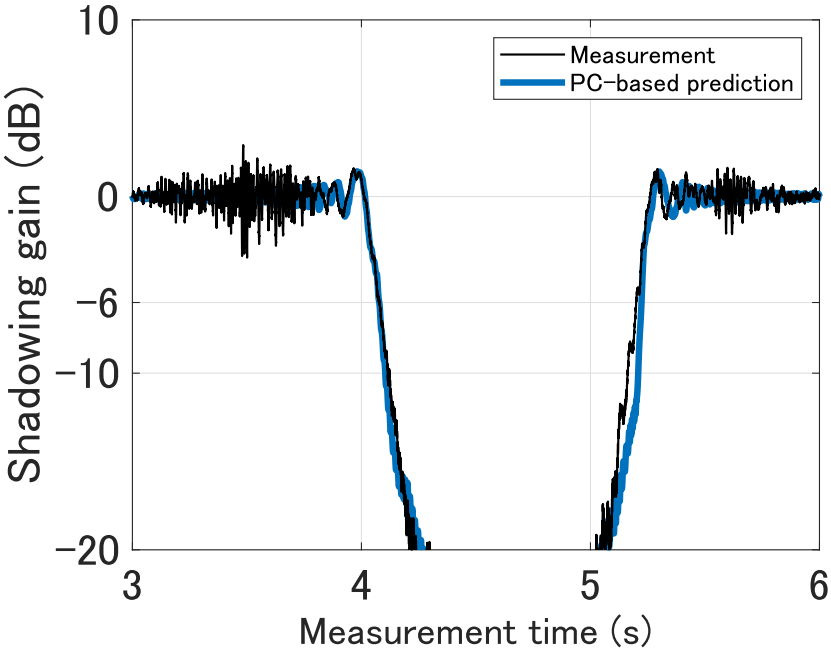

Figure 3.14: .

(MAE) and the worst error must be less than half a frame and a frame, respectively, to guarentee the synchronization accuracy. As a result, the MAE and the worst error were 7.5 ms and 24 ms, respectively. Regarding the sampling rate of the MoCap system, 30 fps, it can be said that the synchronization error of the system is enough to validate the following HBS channel models.

Table 3.3: The time when shadowing gain decays/rises across −6 dB.

	Measurement	UTD simulation
First decay	4.078 sec	4.078 sec
First rise	5.199 sec	5.223 sec
Second decay	9.948 sec	9.949 sec
Second rise	10.859 sec	10.869 sec
Third decay	13.749 sec	13.745 sec
Third rise	14.583 sec	14.597 sec
Mean absolute error	–	7.5 msec
Worst	–	24 msec

3.6 Summary

To develop and validate the PC-based deterministic channel model, the synchronized system consisting of a channel sounder and a MoCap system was developed. The synchronization was achieved by introducing an external trigger to the Rx unit and the MoCap system. The developed synchronized measurement system was validated by comparing the simulated and measured results of a canonical problem, knife-edge diffraction. As a result, the MAE error was found within one-third of a frame interval of the depth camera, which is tolerable for validating the PC-based deterministic HBS channel model. The developed measurement system is available for validating the following PC-based models, i.e., objective 1 has been achieved.

Chapter 4

Point Cloud-based Vertically Long Models

4.1 Introduction

Considering a steady pedestrian motion across the LoS, in the indoor scenario where the LoS path length is rarely longer than 10 m, the first Fresnel radius does not exceed 12 cm, which is small enough compared with a human's chest and arm. Thus, as shown in Fig. 4.1, the 3D shape of the human body can be approximated to a 2D human model, which is defined by the horizontal contour at the height where the LoS is closest to the human body. When the antennas are at the same height, the horizontal contour at the same height as the antennas is used. In this chapter, the processes to generate the proposed PC-based vertically long models, the cross-section screen, the Bullington screen, and the elliptic cylinder model, are proposed. Following the proposals, the experiment validation at 58 GHz is detailed.

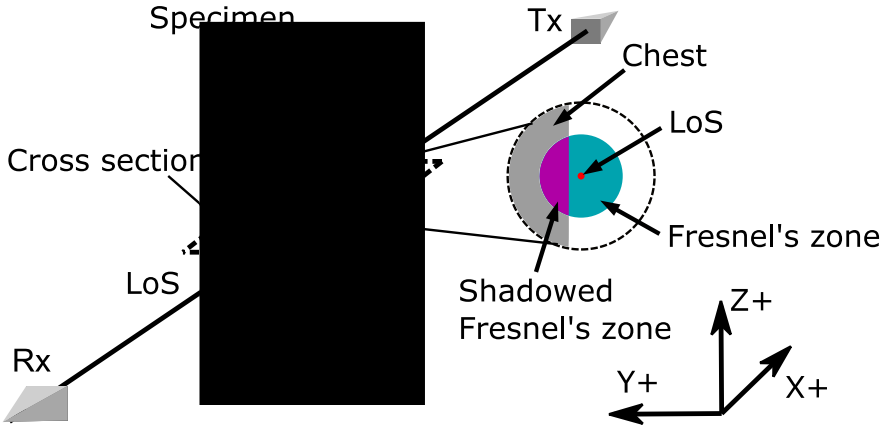

Figure 4.1: The clearance of the Fresnel zone shadowed by the belly of the human body. Because of the short wavelength at the 58 GHz band, the clearance of the Fresnel zone can be considered as a 2D problem.

4.2 Human Models based on Horizontal Contour

Regarding the changing cross-section of the human body walking with a natural gait, we propose three types of PC-based geometric models for the HBS simulation, i.e., the cross-section screen, the Bullington screen, and the elliptic cylinder. The HBS channel simulations are conducted with the proposals and the UTD models introduced in Section 2.2. Among the three proposals, the cross-section screen is superior in computational simplicity, the Bullington screen considers the complex shape of the human body by visibility analysis, and the elliptic cylinder model takes the reflected wave from the human body into account.

4.2.1 Cross-section Screen Model [1, 2]

With the MoCap system synchronized to the channel sounder, the time-varying shape of the human body such as swinging arms can be taken into account. Here, we propose the

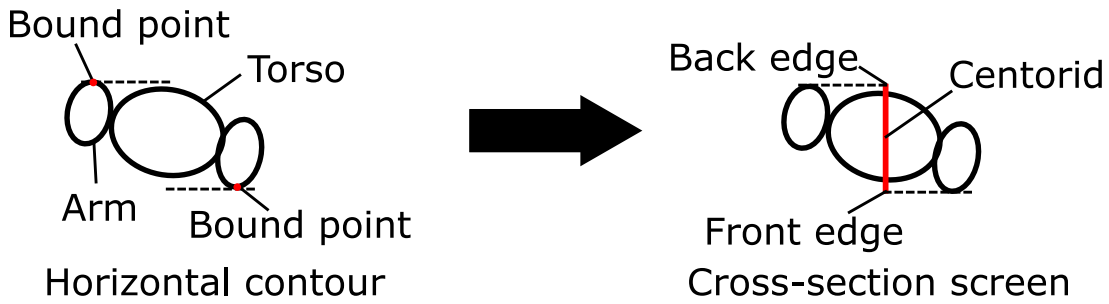

Figure 4.2: The process generating the Cross-section screen model. The coordinates in the direction perpendicular to the LoS of both edges are the same as the bound points in the same direction. The coordinates in the LoS direction are the same as the contour's centroid.

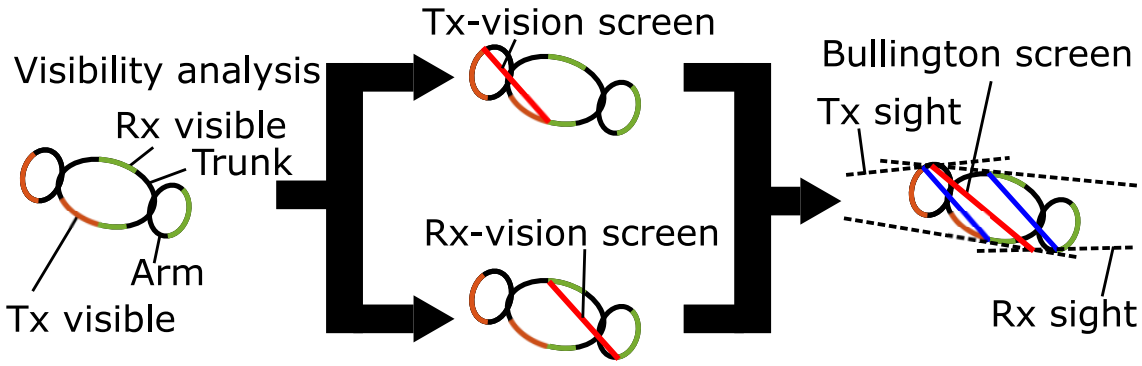

Figure 4.3: The process of generating the Bullington screen model. Firstly, the Tx-vision and the Rx-vision screens are defined by finding the bound points of the points visible to each antenna in the direction perpendicular to the LoS. Inspired by the Bullington model [10], the Bullington screen's edges are found at the intersection of the line drawn from both antennas to the edges of the two screens mentioned above.

cross-screen screen defined by the instantaneous circumference of the cross-section of the human body, as shown in Fig. 4.2. Keeping the face normal to the LoS, the position of the screen along the LoS is the same as the centroid of the horizontal contour of the human body. Positions of the edges are found as the bound points of the contour cloud in the same direction.

4.2.2 Bollington Screen Model

Considering the edges of the cross-section screen may not be visible from the TRx due to the complex shape of the human body, the Bullington screen is defined based on the visibility of the contour points from the viewpoint of the TRx. First, the Tx-vision and Rx-vision screens are defined, as shown in Fig. 4.3. Positions of the edges of the Tx-vision screen and the Rx-vision screen are the positions of both bound points of the visible PC of the 2D contour along the direction perpendicular to the LoS. Then, inspired by the Bullington model [10], the Tx-vision screen and the Rx-vision screen are synthesized into the Bullington screen. The positions of the Bullington screen's edges are defined as the intersection of the lines drawn from the antennas to the edges of the Tx-vision screen and the Rx-vision screen.

4.2.3 Elliptic Cylinder Model

Solid models such as circular and elliptical cylinders are used in [6, 1] to consider the effects of the interference due to the surface of the human body such as the creeping wave and the reflected wave, which are ignored by applying the screen models with the UTD method. Before and after the shadow event, the reflected waves from the surfaces additionally contribute to the received field [49], which enhances the small-scale fading. In the shadow region,

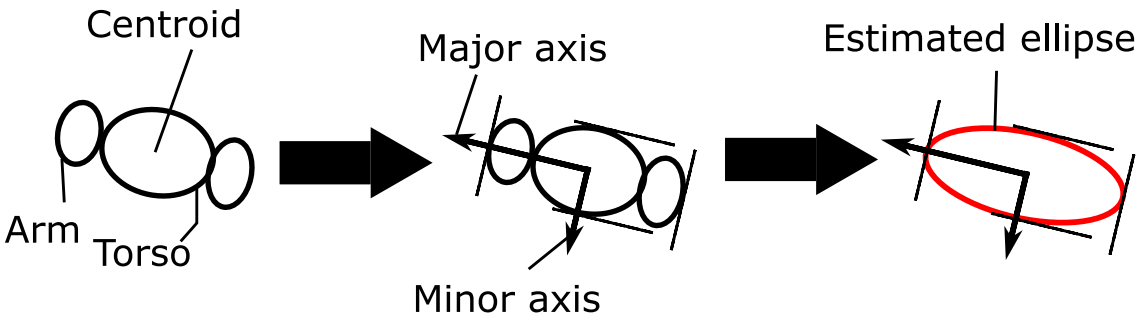

Figure 4.4: The process generating the elliptic cylinder model. The ellipse is defined by applying the principal component analysis to the contour related to its centroid. The major and the minor axes are found as the first and the second principal components. The sizes of both axes are found as the range of the points in the direction of both axes.

the diffraction waves creeping along the surfaces lose more energy than diffracting around a wedge and lead to a deeper shadowing loss [31]. Compared with a circular cylinder, an elliptical cylinder is more similar to the shape of the human body and considers the direction of the human body in addition. The directions, sizes, and positions of the elliptic cylinder model are found by applying the principal component analysis (PCA) to the 2D horizontal contour related to the centroid of the contour, as shown in Fig. 4.4.

4.3 Dynamic HBS Channel Sounding at 58 GHz

To validate the proposed PC-based geometric models of the human body, the results of the propagation simulation using the proposals were compared with the measurement result in an indoor dynamic HBS scenario at 58 GHz band. A simple indoor measurement scenario of a dynamic HBS channel was considered, where a device-to-device (D2D) propagation link was shadowed by a pedestrian walking naturally. The measurement was conducted in an empty meeting room at Niigata University. Since the sounding signal was a CW, the interference of the MPCs from the environment was excluded by using the narrow beam and deploying absorbers behind the antennas. The set-up diagram is shown in Fig. 4.5. The Tx and the Rx were deployed at a height of 1.1 m and separated by 6.5 m. In the measurement, the specimen traversed a linear trajectory across the middle of LoS between the Tx and Rx for 1.5 round trips. The first Fresnel radius at the center of the LoS was 85 mm. The thickness of the specimen at 1.1 m height was about 270 mm. The four cameras of the MoCap system were deployed in the four corners where the complete surface of the human specimen can be recorded.

Using the definitions as shown in Fig. 4.6, The specification of the specimen is identified in Table 4.1. In [50, 51], it was confirmed that even the 1 mm thick cotton material contributes attenuation less than 3 dB at mmW band (up to 300 GHz). Thus, the clothing, a thin

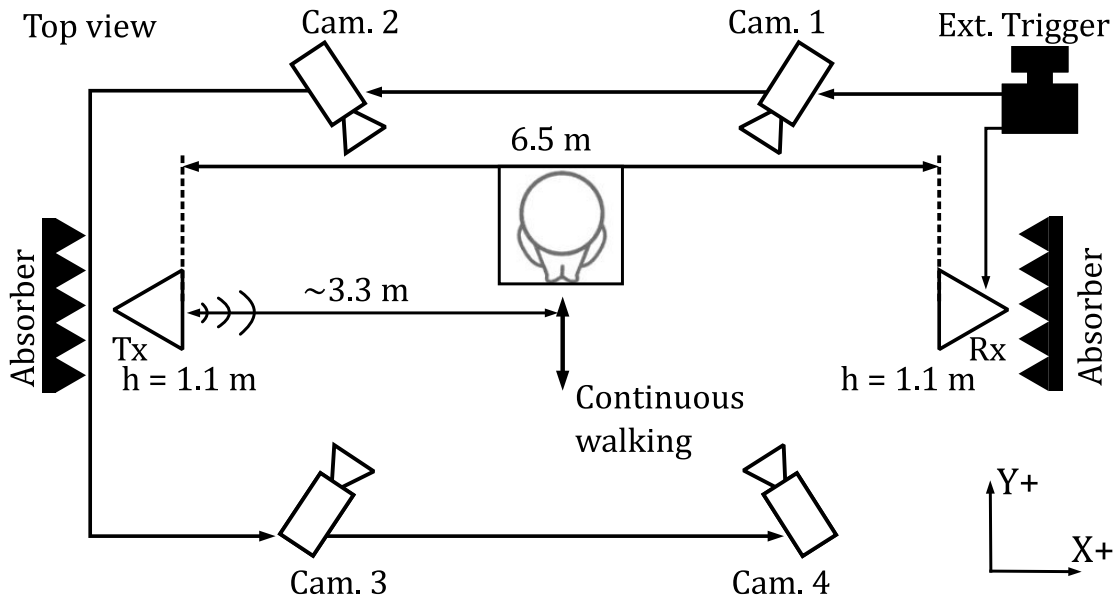

Figure 4.5: Diagram of measurement set-up with human traversing a linear trajectory cross to the LoS between the Tx and Rx.

Table 4.1: Specification of the specimen.

Human subject	Male A
Height	1.72 m
Body depth	0.27 m
Body width (torso)	0.32 m
Body width (arms included)	0.54 m
Cloth	Thin cotton (0.4 mm)

cotton shirt, in the measurement of this work is considered transparent at 58 GHz. Since the clothing is visible at the IR band, the gap between the clothing and the skin of the specimen may lead to an overestimated surface.

The coverage of the HPBW of the RX antenna beam pattern at the specimen was 0.34 m in the horizontal plane and 1.09 m in the vertical plane. Therefore, the multipath fading caused by the diffracted waves at both sides of the specimen could be measured, while the floor-reflected wave and the diffracted wave at the head of the specimen could not be identified.

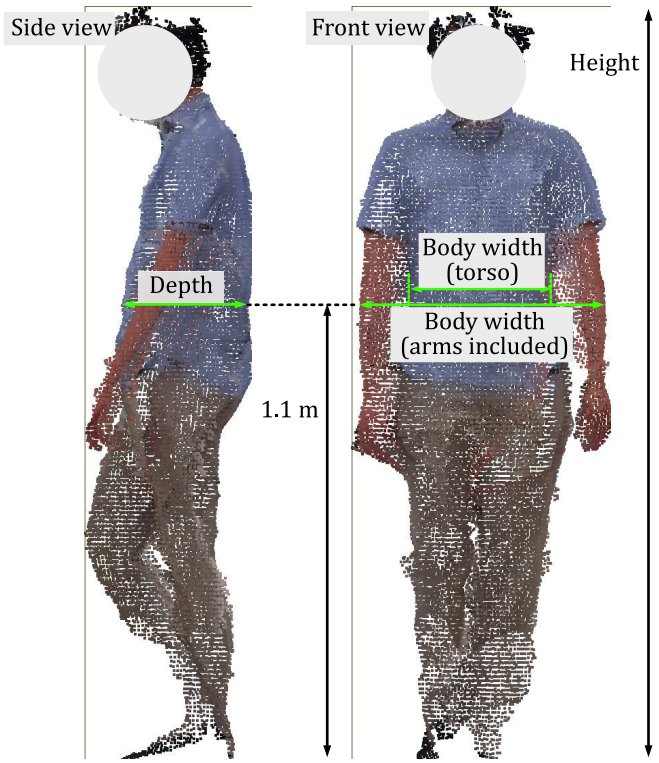

Figure 4.6: The photograph of the specimen with the body dimensions as defined in Table 4.1.

4.4 Results and Discussion

Precise localization and capture of the shape of the human body enable a fair comparison between the proposed PC-based human models and the conventional screen model. By using the synchronized system consisting of the channel sounder and the MoCap system, the screen models and the elliptic cylinder model can be generated from the instantaneous human shape, and the conventional screen model can be localized precisely. As mentioned in Section 3.4, the possible overestimation of the dimension of the surface of the human body was 24 mm in maximum. In this section, we compare the common fixed screen model using premeasured dimensions of the human body with the proposed screen models and the elliptic cylinder model. The definitions of the metrics are shown in Fig. 4.7. First, the measurement result and the predictions of the shadow distance are compared quantitatively. The shadow distance is defined as the distance between the two most prominent peaks on both sides before and after the shadow event for judging when to switch the communication link. The fading distance is defined as the distance between the position of the human body where the shadowing gain drops below the threshold and recovers to indicate how long the communication link is lost. The threshold for the fading distance is defined as -6 dB, which is the shadowing gain of the shadow boundary obtained by letting the edge of a semi-infinite screen transverse to the direction of propagation. Next, the agreement of the fading depth and the interference ripples in the lit region are compared. The results of the three MoCap

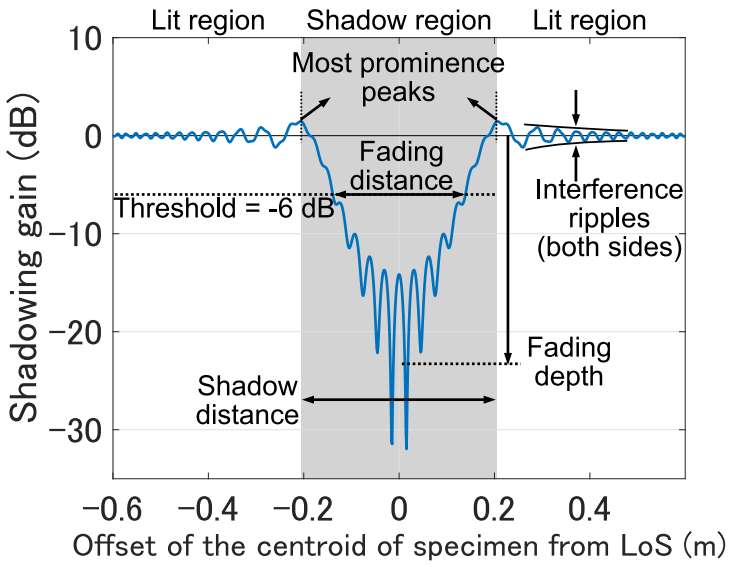

Figure 4.7: The example of the metrics of the propagation channel shadowed by a fixed screen. The shadow distance is defined as the distance between the two most prominent peaks on both sides.

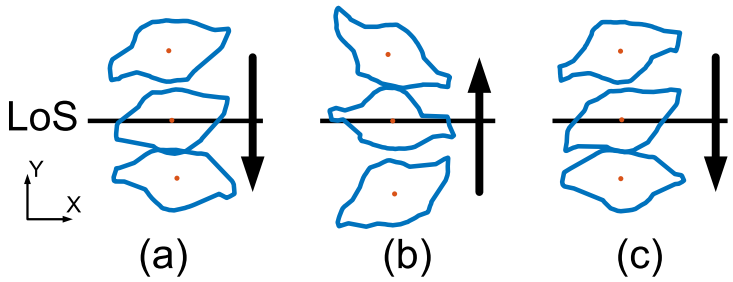

Figure 4.8: The horizontal contour of the specimen in the three measurements. (a) The first outward trip. (b) The return trip. (c) The second outward trip.

measurements are shown in Fig. 4.8. In the outward trips, the right arm approached the LoS first while the torso left the LoS last. On the return trip, the torso approached the LoS first while the right arm left the LoS lastly.

4.4.1 Comparison between the Models against the Shadow Distance

The shadow distance in the HBS event can be used to judge when to switch the communication link to the alternative path and when to resume. Comparing the screen defined by the premeasured body depth in Table 4.1 with our proposed screen models and the elliptical cylinder model, we obtain the results in Table 4.2. In all measurements, the proposed PC-based models outperformed the fixed screen. Compared with the conventional fixed screen model, the MAE of the shadow distances were improved by 51 %, 41 %, and 75 % by using

Table 4.2: The shadow distances of the measurement and the predicted results of the dynamic shadowing channel.

Number of trial	1	2	3	MAE
Measurement	470 mm	511 mm	466 mm	-
Fixed screen	413 mm	413 mm	413 mm	69 mm
Cross-section screen	438 mm	463 mm	444 mm	34 mm
Bullington screen	426 mm	454 mm	444 mm	41 mm
Elliptic cylinder	470 mm	539 mm	489 mm	17 mm

Table 4.3: The fading distances of the measurement and the predicted results of the dynamic shadowing channel.

Number of trial	1	2	3	MAE
Measurement	243 mm	348 mm	277 mm	-
Fixed screen	279 mm	279 mm	279 mm	34 mm
Cross-section screen	273 mm	294 mm	300 mm	36 mm
Bullington screen	275 mm	298 mm	292 mm	32 mm
Elliptic cylinder	340 mm	388 mm	371 mm	77 mm

the proposed Cross-section screen, Bullington screen, and elliptic cylinder models, respectively. The proposals succeeded in following the motion of the swinging arm to recognize the cross-section of the human body, which is approaching the first Fresnel zone. In addition, the screen models tended to make false negatives of the shadow event prediction, while the elliptic cylinder model tended to make false positives of the shadow event prediction. Regarding false negatives causing unexpected communication cut-offs, but false positives only degrade the networking efficiency, the proposed elliptic cylinder model is recommended for the proactive scheduling application.

4.4.2 Comparison between Models against the Fading Distance

The fading distance in the HBS event shows when the communication link is cut off. The results are obtained in Table 4.3. Though the improvement of the MAE of the fading distance due to the proposals was not obvious, the predictions using the proposed screen models can follow the changes between the different measurements. On the other hand, the predictions of the elliptic cylinder tend to overestimate the fading distance. When emulating whether

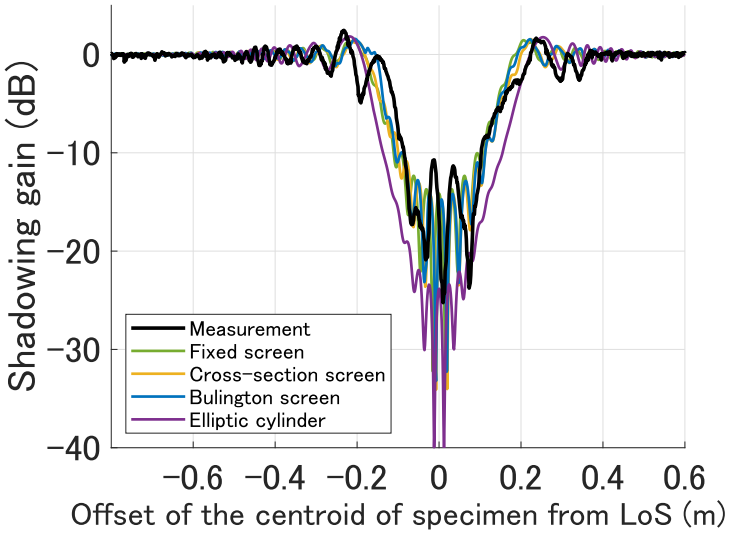

Figure 4.9: The synchronized shadowing gain of the first outward trip.

the propagation link is cut off by the human body, the proposed cross-screen model and the Bullington model are recommended.

4.4.3 Comparison between Models against the Fading Patterns

The strengths of the creeping wave and the scattering wave from the human body are important to simulate the instantaneous shadowing gain accurately. The accuracy of the predicted creeping wave can be evaluated by observing the agreement of the fading depth, and the accuracy of the scattering wave can be evaluated by observing the agreement of the interference ripple in the lit region. The agreement of the fading depth and the interference ripple were observed from the comparison of the patterns of the shadowing gain between the prediction and the measured results, as shown in Figs. 4.9, 4.10, and 4.11.

When observing the fading depth, the elliptic cylinder model overestimated the fading depth of the outward trips, which the screen models predicted precisely. However, the fading depth of the return trip was correctly predicted by the elliptic cylinder model but underestimated by the screen models. The reason was the disagreement between the ellipse and the horizontal contour. As shown in Fig. 4.4, there are unmatched surfaces in the gap between the arms and the torso. When the arm approached the LoS first or left the LoS last, the filled surface at the position where there was a gap blocked the Fresnel zone and contributed to a faster decay or a later rise. On the other hand, by observing the horizontal contours in Fig. 4.8, we can find the disagreement of the fading depths was due to the poor estimation of the creeping length. The creeping lengths along the belly and the back had good agreements to the creeping lengths along an ellipse only when the front direction of the human body was perpendicular to the LoS. In the outward cases, the creeping lengths were

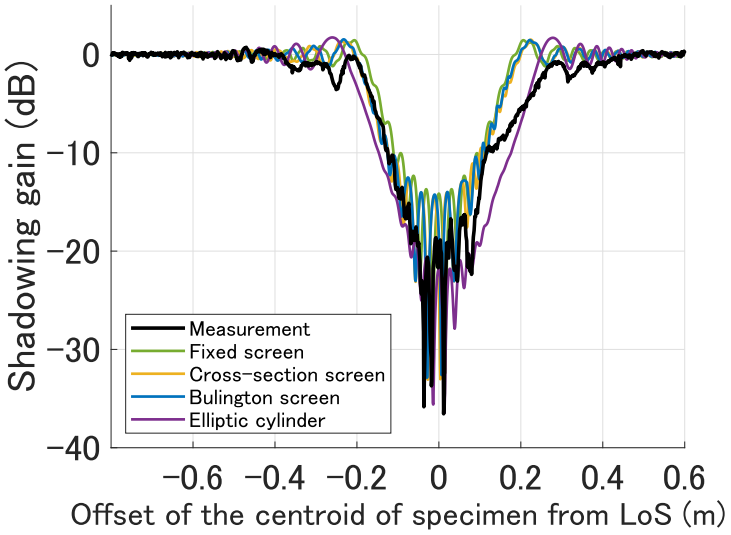

Figure 4.10: The synchronized shadowing gain of the return trip.

overestimated. Since the creeping length contributes to an evanescence term in the UTD calculation in (2.25), the fading depths were overestimated by the elliptic cylinder model in the two outward trips. The results showed a complex model rather than a simple elliptic cylinder model is needed to predict the creeping wave accurately.

In the decay and rise phases between the LoS and the NLoS conditions, where the centroid of the human body was located between −0.3 to 0.1 and 0.1 to 0.3 , the screen models predicted the HBS shadowing gain accurately in most cases. However, there was a significant deviation observed especially in the rising phase of the return trip where the centroid of the body was between 0.1 to 0.3 m, as shown in Fig. 4.10. To investigate the cause of such occasional prediction errors, the difference between the screen model and the instantaneous posture of the human body when the deviation was exhibited should be observed, as shown in Fig. 4.12. It was observed that the torso and the right arm were similarly distanced from the LoS and formed a double-diffraction condition. Since additional diffraction contributed to exceeding attenuation, the screen models overestimated the shadowing gain. Such occasionally overestimated shadow gain caused false negatives observed in Section 4.4.1.

In the observation of the interference ripples in the lit region, it was found the predictions based on the elliptic cylinder model were more accurate compared with the predictions based on the screen models. It showed the scattering wave from the surface of the human body can be modeled as a specularly reflected wave considering the curvature of the surface. Since the discrepancies remained, it suggested that a complex model rather than a simple elliptic cylinder model is needed to predict the reflection wave from the human body accurately.

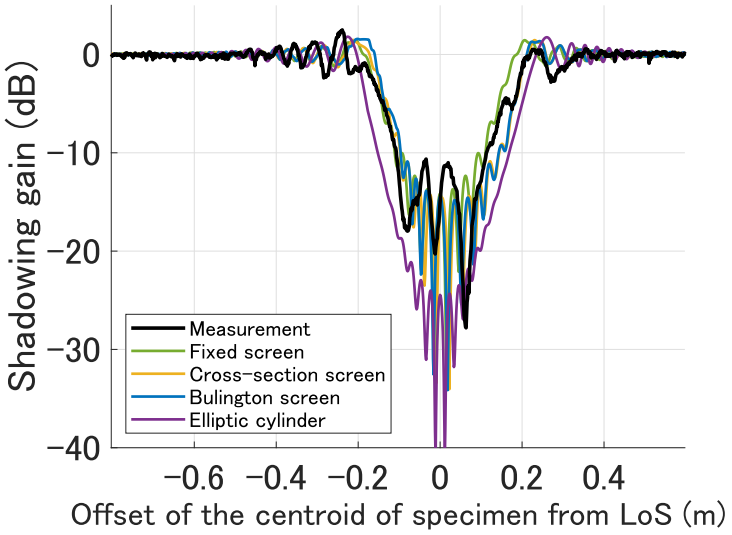

Figure 4.11: The synchronized shadowing gain of the second outward trip.

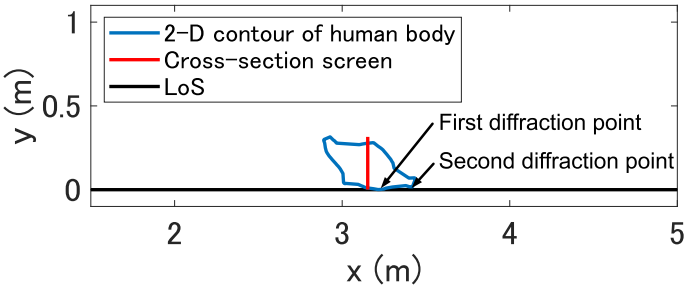

Figure 4.12: The horizontal contour of the human body and the estimated cross-section screen when the centroid of the human body was at 0.17 m in the return trip.

4.5 Summary

This chapter proposed three types of PC-based geometric models that fit the instantaneous human body for the deterministic HBS channel model in the indoor (6.5 m) scenario at 58 GHz. Unlike the conventional fixed screen model, the proposals flexibly change their shape and direction to fit the instantaneous cross-section of the obstacle from the antennas' viewpoint. The flexible models took the detailed geometry of the human body into account and enabled the simulation of the deterministic channel propagation affected by a human with a changing cross-section. To accurately evaluate the proposals, a measurement of the dynamic propagation channel at the 58 GHz band affected by the human body using our synchronized channel sounder and MoCap system was conducted. The shadow distance, fading distance, and the pattern of the shadow gain of the prediction were compared with the measured results. The proposed cross-section screen and the Bullington screen performed well in tracking the change in the fading distances between the measurements. The proposed

elliptic cylinder model improved the prediction of the shadow distance by 75 % on average and the scattering waves from the human body. The results showed the proposed PC-based human body model is effective for the dynamic HBS channel simulator. By limiting the calculation to the 2D coordinate system, the proposed vertical long models can efficiently simulate the HBS channel with improved accuracy, i.e., objective 2 has been achieved.

Chapter 5

Point Cloud-based Diffraction Path Extraction

5.1 Introduction

In Chapter 4 [1, 2], the vertically long human models have been developed. However, the conventional 2D models are limited to the condition that the diffraction paths exist on the sides of the human body. In such conditions, the cross-section surrounding the diffraction point on the side of a human body is similar to a vertical edge. Discrepancy is exhibited when the condition is not fulfilled. In scenarios such as office and aircraft cabin [52], the 2D models even fall undefined when the human obstacle blocks the LoS from outside of the verticle plane including the Tx and the Rx, as shown in Fig. 5.1. Therefore, the fast fading in the lit region cannot be predicted since the PC-based 2D model is undefined.

To accommodate the HBS channel affected by various human motions, an extension for the PC-based 2D model is needed. In this chapter, the diffraction path extraction method estimating the diffraction paths from the complex human body for the HBS channel simulation is proposed.

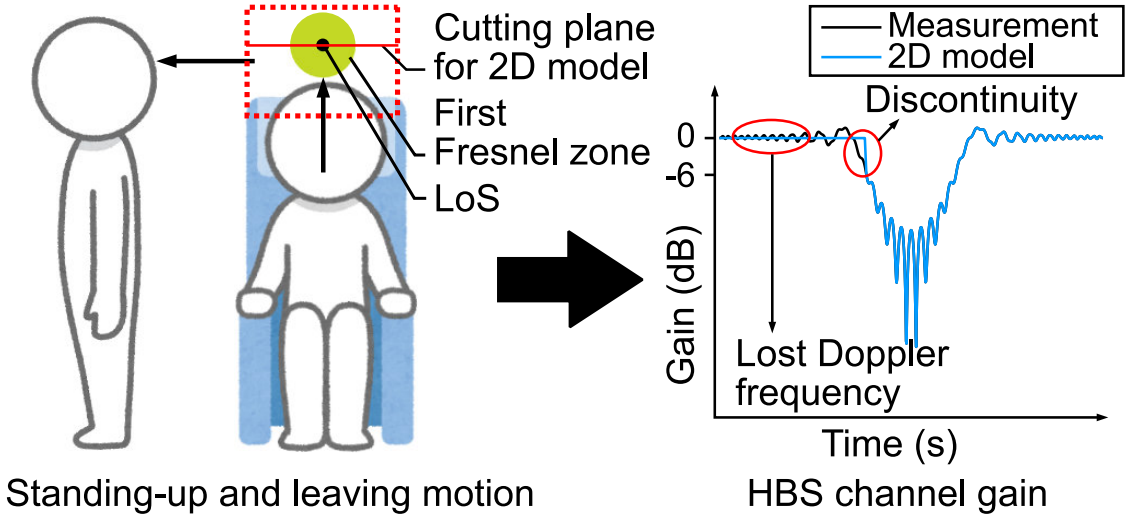

Figure 5.1: The dynamic HBS channel gain related to a standing-up human obstacle in an office scenario.

5.2 Point Cloud-based Diffraction Path Extraction [3]

To recognize the detailed cross-section of the human body from the viewpoints of the antennas, it is proposed to approximate the diffraction waves as the ones from a human body-shaped screen. The body-shaped contour for the simulation is formed by projecting the PC of the human body to the projection plane perpendicular to the LoS. Since the lower body is far from the LoS and contributes little to the HBS channel, the contour of the lower body is formed by connecting to a vertical screen, whose base is on the ground, as shown in Fig. 5.2. The projection plane is located where the representative point of the human obstacle is included. As the joint nearest to the LoS when the LoS is fully blocked in the office scenario, the neck joint is chosen as the representative point. The neck joint of the human obstacle can be found from the human skeleton estimated with the Microsoft Azure Kinect body tracking (BT) software developing kit (SDK) [53]. The machine learning (ML) model of the Microsoft Azure BT SDK estimates the skeleton of the human obstacle from the viewpoint of each depth camera, as shown in Fig. 5.3. Considering the uncertainty of the skeleton estimation, the position of the neck joint of the human model is found as the average point of the visible neck joints estimated from each depth camera.

With the human body-shaped screen, the diffraction paths for the UTD simulation can be extracted. According to Fermat's principle [54, 55], the diffraction paths exhibit local extrema in path length. Since the phase terms of the diffraction points are stationary, the diffraction points are also the stationary phase points (SPPs), as shown in Fig. 5.4. Therefore, the criterion for separating SPPs by the difference of the Fresnel number $\Delta n_{\mathrm{FN}} = 3$ in [28] can be used for separating the diffraction points. With the extracted diffraction paths and the UTD method, the scattering from the human obstacle is simplified to N_{SPPs}

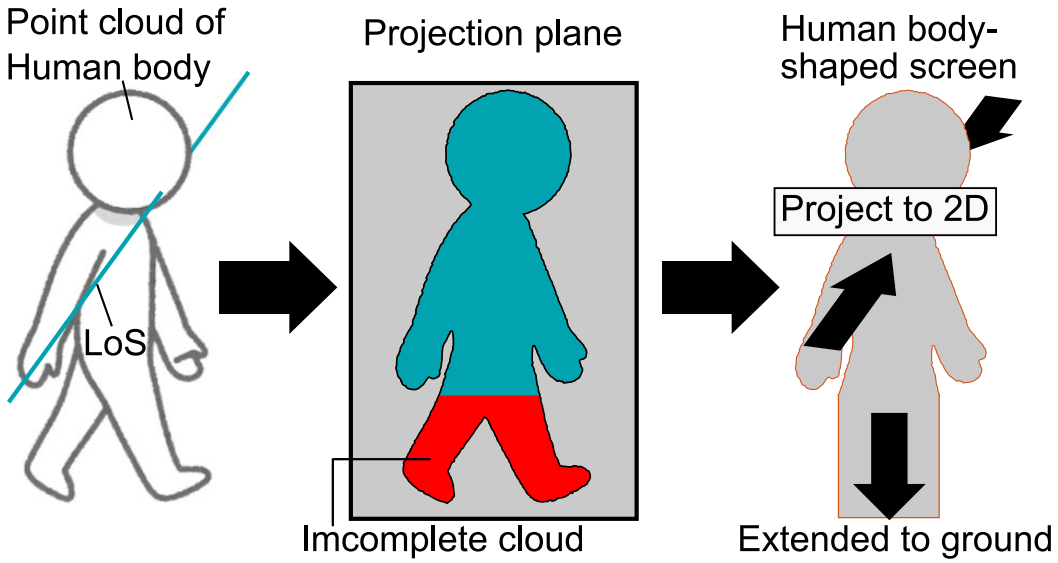

Figure 5.2: The process of generating the human body-shaped screen model. The shape of the screen is the contour of the PC projected to the vertical projection plane. The coordinates in the LoS direction are the same as the centroid of the horizontal contour.

diffraction from absorbing edges. The direct path exists when the LoS does not interact with the human body-shaped screen. Using the UTD method, the received field is calculated as the summation of the contributions from all the paths as

$$\boldsymbol{E}_{\mathrm{DP}}^{\mathrm{Rx}} = \begin{cases} \sqrt{G^{\mathrm{Tx}} G^{\mathrm{Rx}}} \boldsymbol{E}^{\mathrm{LoS}} \cdot \boldsymbol{u}_{\mathrm{Rx}} + \boldsymbol{E}_{\mathrm{DP}}^{\mathrm{D}} \cdot \boldsymbol{u}_{\mathrm{Rx}}, & (\mathrm{LoS}) \\ \boldsymbol{E}_{\mathrm{DP}}^{\mathrm{D}} \cdot \boldsymbol{u}_{\mathrm{Rx}}, & (\mathrm{NLoS}), \end{cases} \tag{5.1}$$

$$\boldsymbol{E}_{\mathrm{DP}}^{\mathrm{D}} = \sum_{i=1}^{N_{\mathrm{SPP}}} \boldsymbol{E}_{i}^{\mathrm{D}} \tag{5.2}$$

where $\boldsymbol{E}_{\mathrm{DP}}^{\mathrm{D}}$ is the summation of all the fields from the extracted diffraction paths. The

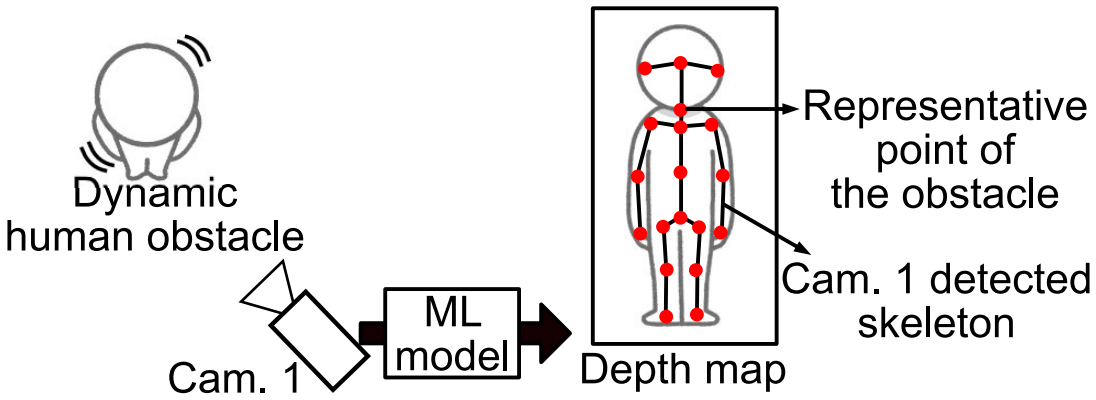

Figure 5.3: The representative point of the human obstacle found as the neck joint with ML model.

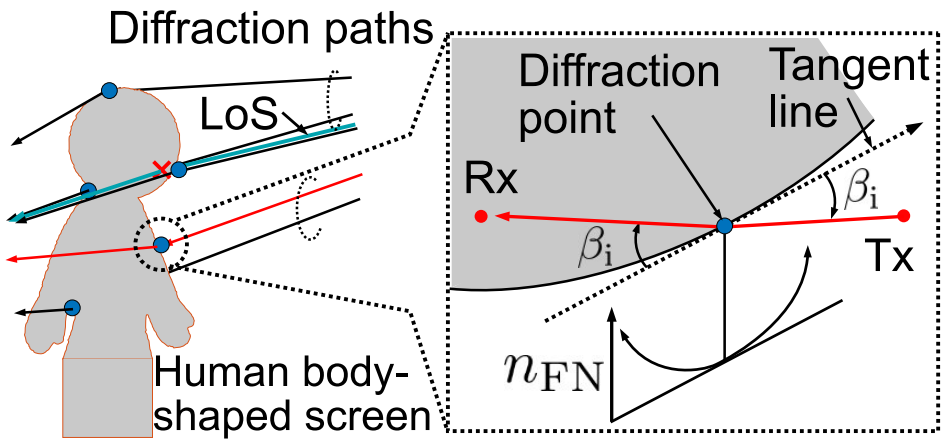

Figure 5.4: Diffraction paths extracted from the human body-shape screen model.

individual field of each diffraction path $\boldsymbol{E}_i^{\mathrm{D}}$ is calculated based on the UTD method by (2.8) mentioned in Section 2.2.2.

Using the combination of the UTD method and the extracted diffraction paths, the HBS channel can be simulated with flexibility in complex human motions.

5.3 Dynamic HBS Channel Sounding at 300 GHz

A synchronized dynamic HBS channel measurement and MoCap at 300 GHz in an indoor scenario were considered. The measurement was conducted in the office of Tokyo Institute of Technology. The set-up diagram of the HBS channel measurement in the office environment is shown in Figs. 5.5 and 5.6. The MPCs from the environment were excluded by using the narrow beam antennas. The Tx and the Rx were set at where the AP and the laptop computer are typically located, the beam and the seat beside the table, where the TRx heights were 2.25 m and 0.76 m, respectively, and the antenna separation was 2.97 m. Regarding the PC-represented human body considered the difference between the various shapes of different specimens, there was one human specimen in the measurement. Acted as the person sitting on the other side of the Rx antenna, the trajectory of the human obstacle was at the seat across the table. In the measurement, the human specimen took and left the seat between the antennas, as one of the typical motions observed in the office. From the viewpoint of the Tx antenna, the human obstacle accessed the seat on the right-hand side. Two directions of the human body during the motion, toward the Rx and right-hand side, were considered. The measurement of each motion was conducted ten times. The first Fresnel radius at the intersection of the LoS and the route was 29 mm. The specifications of the measurement are identified in Table 5.1.

In the MoCap measurement for human motion, the four depth cameras were deployed in the four corners where the complete surface of the human specimen can be recorded. The

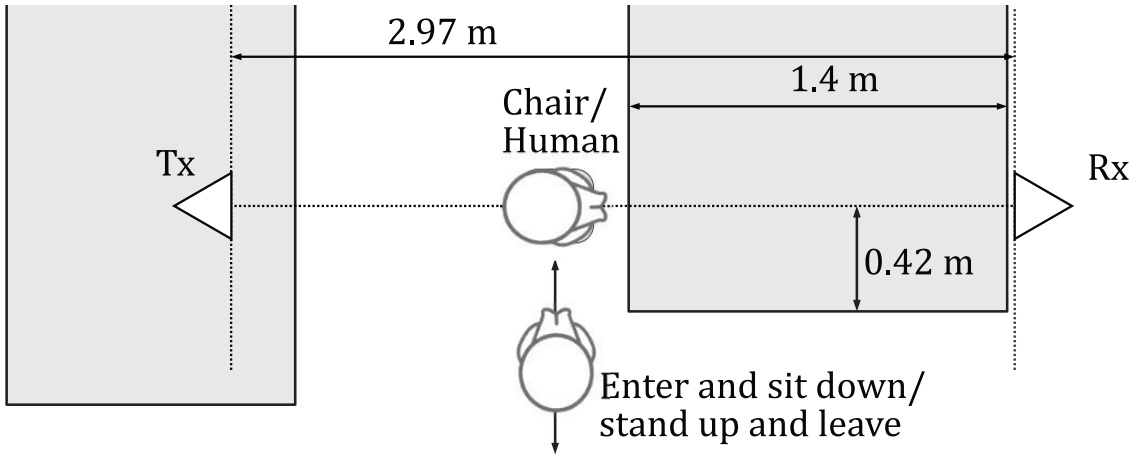

Figure 5.5: Top view of measurement set-up with a human obstacle taking and leaving a seat in an office scenario.

Table 5.1: Specification of the synchronized MoCap and HBS channel measurement.

Number of the human specimen	1
Antenna separation	3.27 m
Antenna tilt θ_{tilt}	25°
Motions	Sitting-down, standing-up
The direction of the human specimen	Front, right-hand side
Number of experiments	10 times per direction
Recording period	15 sec

surface of the head can not be measured by the LiDAR-based depth cameras since the IR light can be absorbed by the black hair. To address this issue, a hood was used to make the surface measurable, as shown in Fig. 5.6. Since the head of the human obstacle was covered by the hood in the measurement, the effect of the clothing on the channel at 300 GHz should be considered. In [50, 51], it was confirmed that the 1 mm thick cotton material contributes attenuation of less than 3 dB at 300 GHz. Thus, the clothing, a thin cotton hood, in the measurement of this work is considered transparent at the 300 GHz band. Considering the clothing is visible at the IR band, the discrepancy between the radio environment and the geometry measured by the MoCap system may exist. To address this, the hood was held tight to minimize the gap between the hood and the skin.

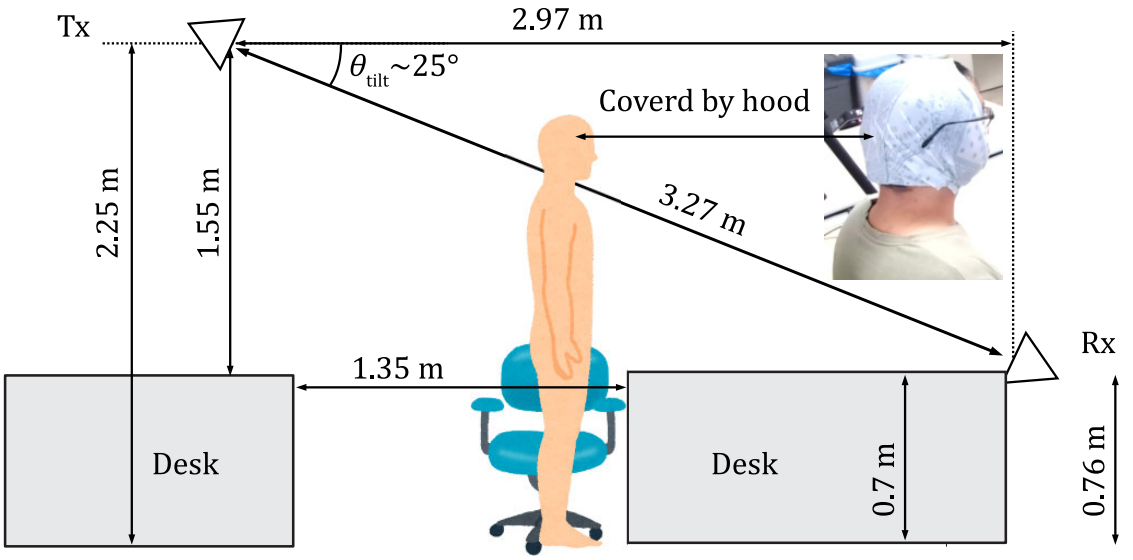

Figure 5.6: Side view of measurement set-up with a human obstacle taking and leaving a seat in an office scenario.

5.4 HBS Channel Simulation in the Lit Region

The sampling rate of the PC-based simulations is limited by the frame rate of the MoCap system, i.e., 30 frames per second (fps). In addition, the maximum Doppler frequency $\nu_{\max}$ is limited to 157 Hz due to the narrow HPBW of the antennas. Therefore, the Doppler frequencies caused by the approaching/leaving human obstacle cannot be observed clearly. To validate the accuracy of the simulation methods in predicting fast fading in the lit region, a comparison of the simulations is conducted to increase the sampling rate and remove the antenna beam pattern. In the simulation, the PC-represented human body fully blocking the propagation channel was used. The layout of the antennas in the simulation was the same as the measurement as described in Section 5.3. The trajectories of the PC in the simulation were in the directions perpendicular to the LoS, as shown in Fig. 5.7. The two trajectories of the human obstacles were set 500 wavelengths (i.e., 0.5 m) at 300 GHz, from

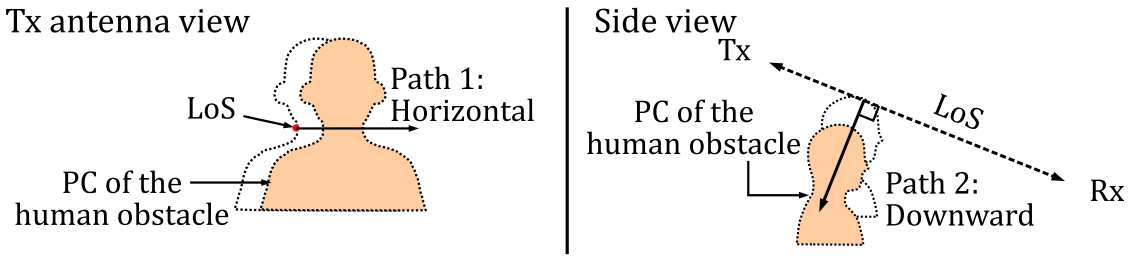

Figure 5.7: Diagram of the simulation set-up. The antennas were deployed the same as the measurement. The human obstacle moved horizontally and downward in the simulation. The ranges of the simulation were 100 wavelengths starting at the shadowing boundary.

the shadow boundary. The moving speed of $v = 1\frac{\mathrm{m}}{\mathrm{s}}$ was assumed and the interval of the sampling points was one-fifth of a wavelength (i.e., 0.2 mm).

5.5 Results and Discussion

With the synchronized system consisting of the channel sounder and the MoCap system, the accuracy of the simulation models can be validated by comparing the simulation results with the measurement results. In this section, the qualitative comparison of simulated shadowing gain against the measurement was conducted to validate the accuracy generally. The following two quantitative comparisons were conducted to validate the accuracy of the simulated shadowing gain in the shadow region and the short-term fading in the lit region.

5.5.1 Comparison in the Shadow Region against the Measurement

The prediction of HBS gain, the inverse value of HBS loss, is used to evaluate the degradation of the occasionally blocked propagation channel. Since the sounding signal was CW, severe fast fading caused by frequency-specific destructive interference was expected. Because of the signal power distributed over the bandwidth, such a fading dip is eased in wideband systems. Therefore, for validating the signal power affected by human blockage, the envelope of the fading pattern should be observed. The envelope of the fading pattern of a narrow band channel is the result of constructive interference. In analytic simulations, the envelope can be calculated as the linearly summed field strength of every diffraction wave $\left|E_{\mathrm{env}}^{\mathrm{Rx}}(t)\right|$, given as

$$\left|E_{\mathrm{env}}^{\mathrm{Rx}}(t)\right| = \begin{cases} \left|E^{\mathrm{LoS}}\right| + \sum_{i=1}^{N} \left|\boldsymbol{E}_i^{\mathrm{D}} \cdot \hat{\boldsymbol{u}}_{\mathrm{Rx}}\right|, & \text{(LoS)} \\ \sum_{i=1}^{N} \left|\boldsymbol{E}_i^{\mathrm{D}} \cdot \hat{\boldsymbol{u}}_{\mathrm{Rx}}\right|, & \text{(NLoS)} \end{cases} . \tag{5.3}$$

In the measurement, the envelope can not be observed due to the discrete samples. Therefore, the envelope should be estimated from the distribution of the measured HBS gain within a moving interval. The distribution of HBS gain can be derived by the two-path model [19, 49] and the random phase approach (RPA) [56]. The two-path model was considered for the HBS channel since the dominance of the propagation channel was the direct wave and one reflection wave in the lit region or two diffraction waves from both sides of the human body in the shadow region. The cumulative probability of the received field strength $P(|E|)$ can be calculated as

$$P(|E|) = \frac{2|E|}{R} \sum_{n=1}^{\infty} \frac{\Phi(\frac{\gamma_n}{R})}{\gamma_n J_1^2(\gamma_n)} J_1\left(\frac{\gamma_n |E|}{R}\right) \tag{5.4}$$

where $J_0(\cdot)$ and $J_1(\cdot)$ are the zeroth-order and the first-order Bessel function, respectively. γ_n is the n-th root of $J_0(\cdot)$. $\Phi(\cdot)$ and R are calculated as

$$\Phi(x) = \exp\left(-\frac{\sigma^2 x^2}{2}\right) \cdot \prod_{i=1}^{2} J_0(A_i x), \tag{5.5}$$

$$R = \sum_{i=1}^{2} A_i + 5\sigma \tag{5.6}$$

where A_i and σ^2 are the amplitude of the i-th ray and the receiver noise power, respectively. A_i should be chosen large enough so that R can be sufficiently larger than σ. With RPA, the ratio of A_1 to A_2 can be found by the 5-th and 95-th percentile of the measured of the received field strength within a moving interval. Finally, the envelope of the received field can be calculated as

$$P_{\text{env}}^{\text{int}}(t) = \big(A_1(t) + A_2(t)\big)^2. \tag{5.7}$$

The moving interval was defined as the uncertainty of the measurement of the human surface within $\pm 2\sigma_{\text{model}}$, i.e., 48 mm. The shadow region is determined with a -10 dB pre-defined threshold of the HBS gain [57]. From the measurement result, the fading depth was found as the median of the moving averaged HBS loss in the shadow region. The median values of HBS loss when the specimen faced front and right-hand directions were found at 22.0 dB and 21.7 dB, respectively.

The accuracy of predicting the HBS gain was validated by observing the logarithmic prediction error ϵ_{S} of the HBS gain in the shadow region as

$$\epsilon_{\text{S}} = 10\log_{10}\frac{\left|E_{\text{env}}^{\text{Rx}}(t)\right|^2}{\left|E^{\text{LoS}}\right|^2} - 10\log_{10}\frac{P_{\text{env}}^{\text{int}}(t)}{P^{\text{LoS}}}. \tag{5.8}$$

The cumulative distribution function (CDF) of the prediction error of ten measurements for the front and right-hand directions is shown in Figs. 5.8 and 5.9. The results were similar regardless of the direction of the human body since the first Fresnel zone was always fully blocked in both scenarios. In terms of the median value of prediction error, the proposal outperformed the conventional vertical screen model by a factor of four. In terms of the distribution, both models performed well in that 90% of the prediction errors fell within a 18 dB range. From the results, it can be said that the proposal is more accurate in accessing the HBS loss of a propagation channel fully blocked by a human body.

5.5.2 Comparison in the Lit region against the Measurement

When the human obstacle approaches the LoS, time-varying scattering waves from the human surface can lead to Doppler frequencies [49]. In wireless systems, the Doppler frequencies-caused fast fading distorts the symbols for quadrature amplitude modulation (QAM), as

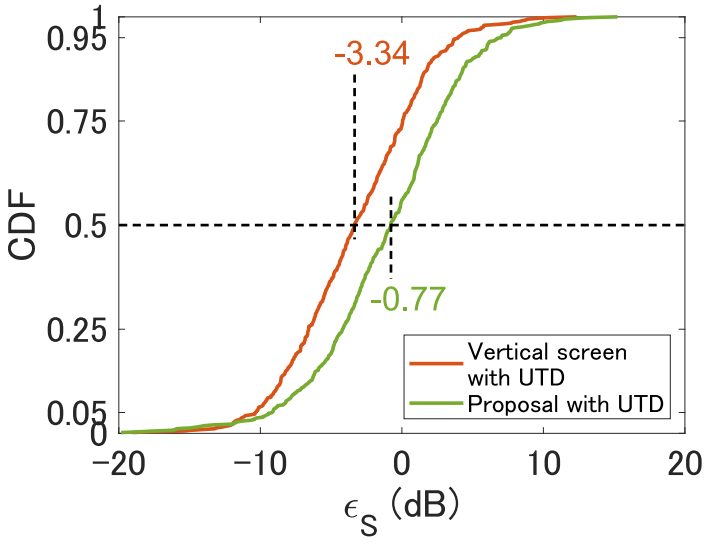

Figure 5.8: The CDF of the prediction error of the HBS gain of the front direction.

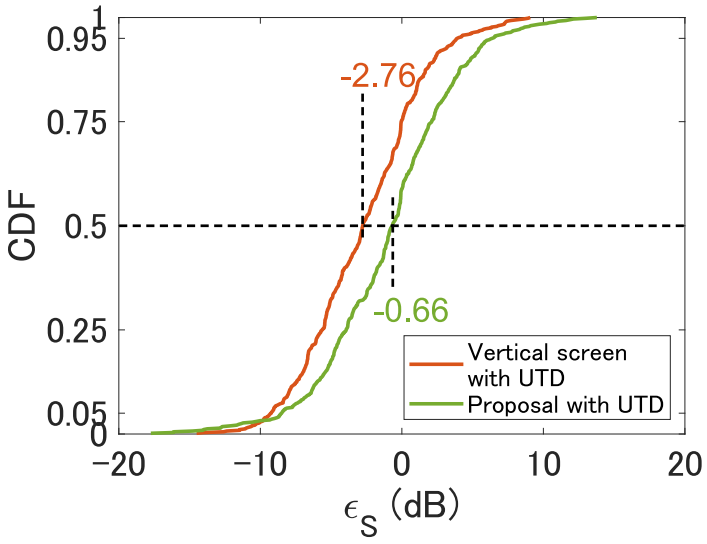

Figure 5.9: The CDF of the prediction error of the HBS gain of the right-hand direction.

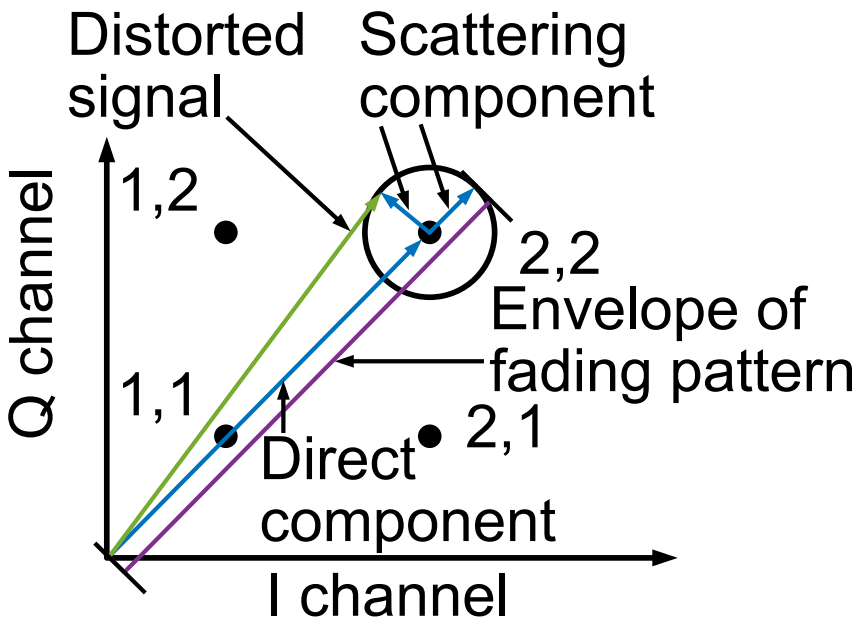

Figure 5.10: The example of the received signal distorted by the scattering wave from the human body.

shown in Fig. 5.10. Due to the sparse sampling rate of the MoCap system and the narrow HPBW of the antennas, the Doppler frequencies in measurement are insufficient for validating the simulations. Alternatively, the fast fading-caused distortion can be observed by the ratio of the envelope of the fading pattern to the direct field (i.e., the dominant component of the HBS channel). From the fading pattern, the envelope can be extracted as the amplitude of the analytic form [58]. From the fading pattern, the envelope can be also extracted by the formulations (5.4)-(5.7).

The accuracy in predicting the envelope is validated by observing the prediction error ϵ_{L}, given as

$$\epsilon_{\mathrm{L}} = \frac{\left|E_{\mathrm{env}}^{\mathrm{Rx}}(t)\right| - \left|E^{\mathrm{LoS}}\right|}{\left|E^{\mathrm{LoS}}\right|} - \frac{\sqrt{P_{\mathrm{env}}^{\mathrm{int}}(t)} - \sqrt{P^{\mathrm{LoS}}}}{\sqrt{P^{\mathrm{LoS}}}}. \tag{5.9}$$

The CDFs of the normalized prediction error of ten measurements for the front and the right-hand directions are shown in Figs. 5.11 and 5.12. The proposal achieved improvement in terms of median error by a factor of five in both directions. The negative bias in the conventional model was because of the ignored or underestimated diffraction waves. Since the obstruction depths of the diffraction paths from the vertical screen are longer than the real ones from the human body, the amplitude of the diffraction wave was always underestimated [6]. Regarding random errors, 90% of the prediction errors fell within a 20% range in both the conventional model and the proposal. Therefore, compared with the conventional model, the proposal is more accurate than the conventional PC-based vertical screen models.

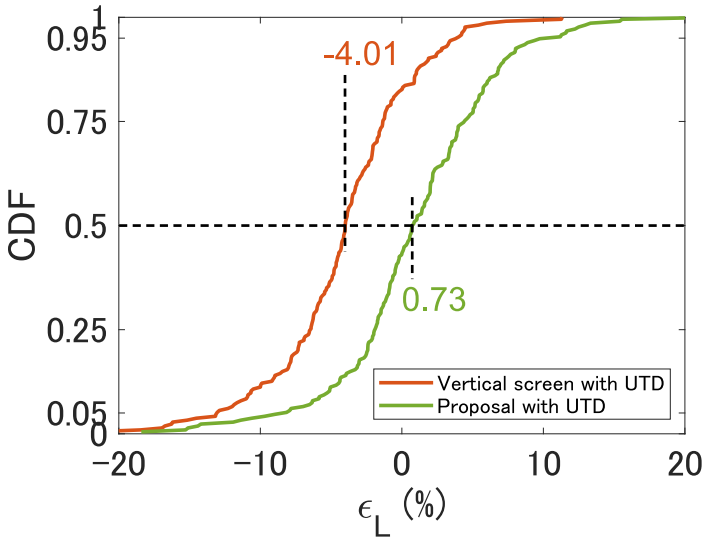

Figure 5.11: The CDF of the normalized prediction error of the envelope in the lit region of front direction.

5.5.3 Comparison of Simulated Doppler Spectra in the Lit Region

To evaluate the Doppler frequencies caused by dynamic human obstacles, simulations can serve as an effective alternative for reference. The sampling rate of a simulation-based reference can be increased easily by using a fixed human obstacle. The MER-EECs method was chosen as the reference considering the complex computation of full-wave methods is intolerable. The simulated time-varying Doppler spectrum is displayed in Figs. **??** and **??**. The Doppler spectrum $P(\nu)$ was calculated as the short-time Fourier transform of the time-varying HBS gain. For the calculation, a time window of 10 ms was used corresponding to a distance of about 10λ to fulfill the assumption of wide-sense stationery. Before the Fourier transform, a Hamming window was applied. According to (**??**) and the geomety, $\nu_{\max} \simeq 600$ Hz. In all the simulations, the proposal matched well with the MER-EECs method. On the other hand, the vertical screen model-based result showed distortion. In the horizontal direction, the Doppler frequencies caused by the scattering waves from the shoulder were ignored. In the downward direction, the Doppler frequencies were not predicted at all since the screen model was undefined. Therefore, the proposal was proved to provide a robust solution to complex human motion for simulation of the fast fading in the lit region. Further experimental investigation is needed to evaluate the scattering component from the shoulders that the proposal failed to predict.

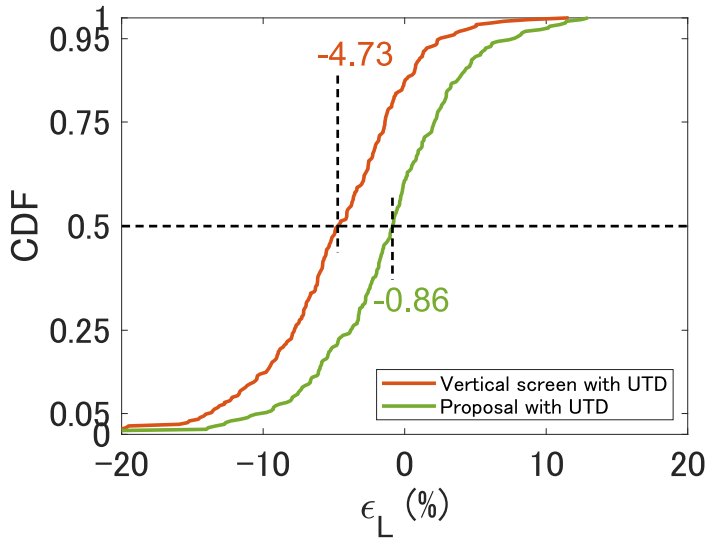

Figure 5.12: The CDF of the normalized prediction error of the envelope in the lit region of the right-hand direction.

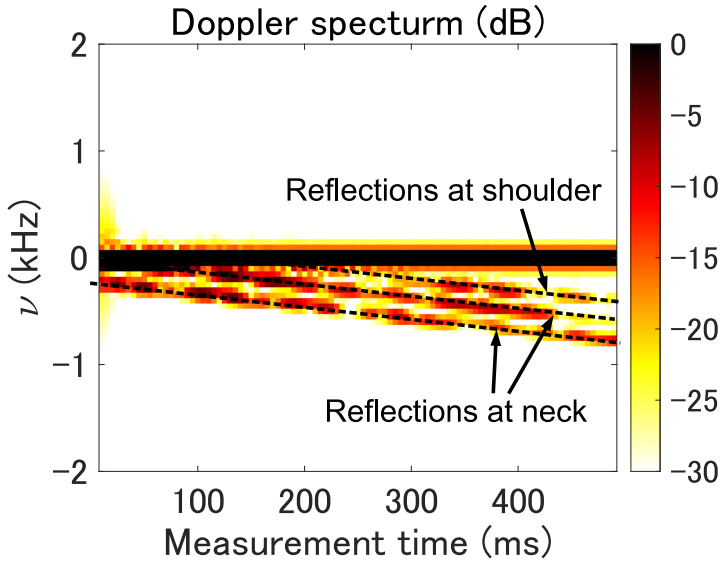

Figure 5.13: The Doppler spectra based on the MER-EECs simulations when the human model moved horizontally.

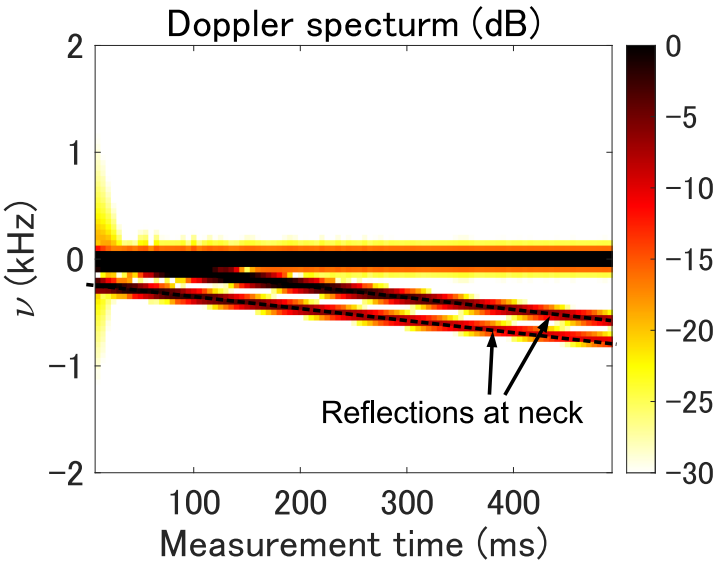

Figure 5.14: The Doppler spectra based on the simulations using the vertically long screen model when the human model moved horizontally.

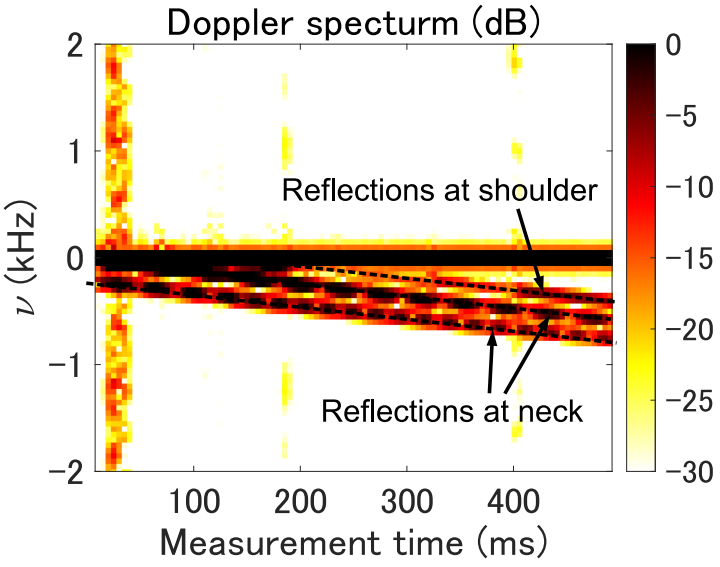

Figure 5.15: The Doppler spectra based on the simulations using the proposal when the human model moved horizontally.

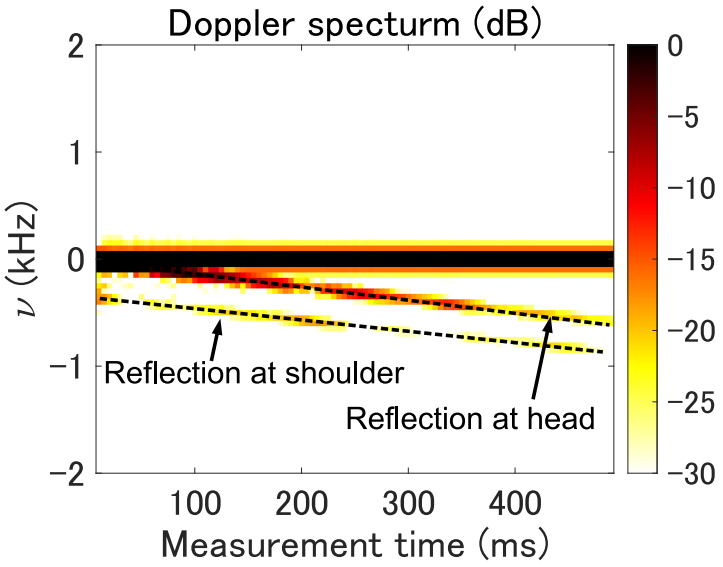

Figure 5.16: The Doppler spectra based on the MER-EECs simulations when the human model moved downward.

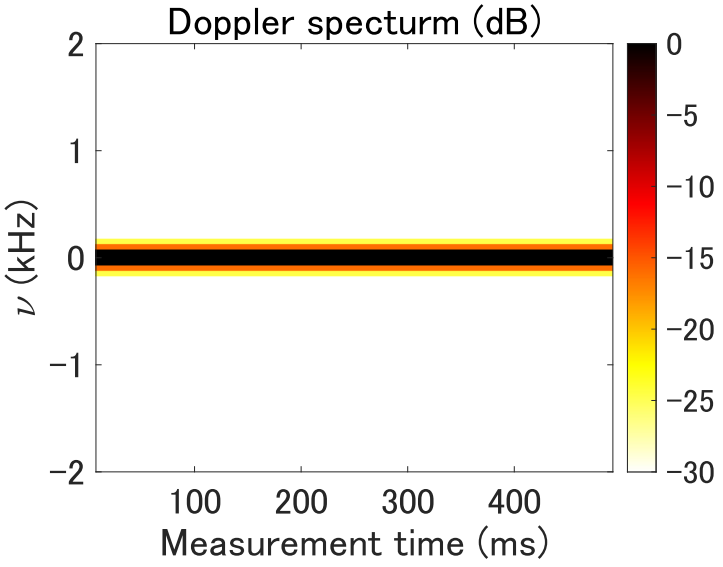

Figure 5.17: The Doppler spectra based on the simulations using the vertically long screen model when the human model moved downward.

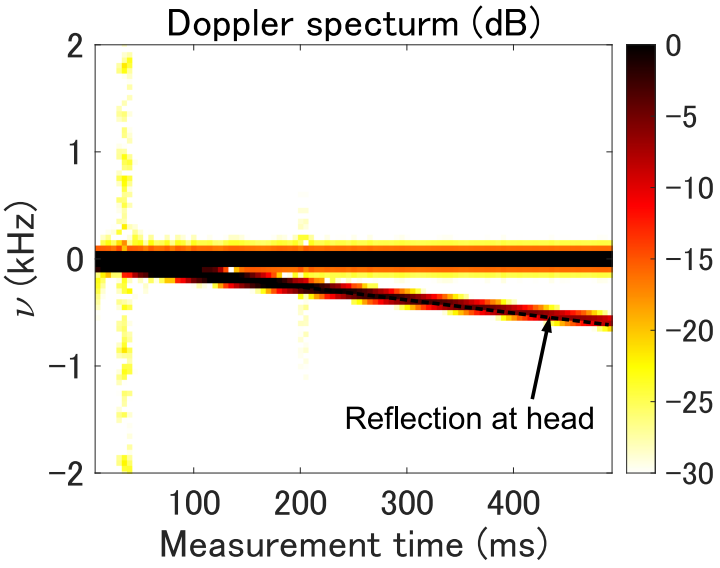

Figure 5.18: The Doppler spectra based on the simulations using the proposal when the human model moved downward.

5.6 Summary

he conventional vertical screen model, the proposal accurately estimates the diffraction paths from the detailed human shape. For validation, twenty measurements of the dynamic HBS channel at 300 GHz and a comparison between simulations were conducted. The human shape corresponding to the instantaneous channel was measured as a PC by the synchronized channel sounder and MoCap system. It was found that the proposal was more accurate in predicting the fast fading in both shadow and lit regions. The proposal achieved an improvement in the median logarithmic error by a factor of four in the shadow region and an improvement in the median linear error by a factor of four in the lit region. Furthermore, the proposal proved a robust solution to complex human motion in the comparison between the simulations. It was proved that when the LoS is expected to be blocked by the upper part of a human body, the diffraction waves can be approximated as the ones from a human body-shaped screen. To summarize, in the simulation for the fading pattern close to the HBS event, the proposal is superior to the PC-based vertically long models in the accuracy and flexibility of the motion of the human obstacle. That is, the proposal serves as a flexible and accurate solution for Objective 2.

Chapter 6

Conclusion

6.1 Summary

This study proposed a synchronized measurement system consisting of a dynamic channel sounder and a MoCap system. The proposed synchronized measurement system can measure the shape of the human body and with corresponding channel response with a dense sampling rate. The synchronized measurement system was validated by a canonical problem, knife-edge diffraction, at 300 GHz. This study also proposed various PC-based human models for ray-based HBS channel simulation at EHF bands. The proposed models were validated with the aforementioned measurement system at 58 GHz and 300 GHz bands. It was found that the vertically long models feature efficiency and good accuracy in the shadow duration with a maximum improvement of $40\ \% - 75\ \%$ compared with the conventional fixed-shaped screen model. However, the vertically long models can not predict the fading pattern affected by a complex human motion accurately. To fill this gap, the diffraction path extraction method can simulate the fading pattern affected by any human motion by sacrificing some efficiency.

6.2 Contribution

This study provides a reliable synchronized measurement system for the evidence of the HBS channel simulation and simulation techniques generated from arbitrary human shapes. In the former, not only the development of deterministic HBS channel models but also the empirical models can benefit from precise and detailed evidence. In the latter, the PC-based models can simulate the HBS channel gain deterministically with the shape of a specific human obstacle with experimental evidence. The results can contribute to the further development of HBS channel simulation using 3D computer graphics-based (3DCG-based) human phantom. Since the Doppler spectrum in the lit region can be used for predicting the shadowing time, the achievement can be included into the recommendation document

dealing with the prediction model for the effect of the human body shadowing problem, which is under discussion by the corresponding group, CG 3k-21, International Telecommunication Union Radiocommunication Sector (ITU-R).

6.3 Applicability for the Future HBS Channel Models

With respect to the development of the empirical models, the detailed shape of the human body captured by the proposed synchronized measurement system can help the generalization with detailed parameters of the motion, age, gender, and race of the human obstacle. With respect to the development of the deterministic models, this work provides evidence linked to the channel measurement. Such evidence helps to validate the HBS channel model using the 3DCG-based human phantom, whose experimental validation is difficult. In the aspect of mobile communication at microwave band, this work may help the development of the short-range dynamic radio channel emulator if experimental validation is conducted.

Appendix A

Manual for the Proposed Synchronized Measurement System

A.1 Introduction

In this chapter, the manual for the experiment using the proposed synchronized measurement system mentioned in Chapter 3 is introduced. In Section A.2, the installation guide is presented. In Section A.2.2, the operation manual is presented.

A.2 Installation guide

In this section, the installation guide for the synchronized measurement system consisting of the channel sounder and the MoCap is introduced. Though there are two channel sounders mentioned in this thesis, only the channel sounder at 300 GHz is introduced. The installation guide for the channel sounder at 58 GHz can be found in [43]. The general steps for installation are described in the following subsections, as follows.

1. Install the dynamic channel sounder.
2. Install the MoCap system.
3. Connect the trigger cable to both instruments.

The checklist of the items is,

- Home-made trigger combo
- Laser measure
- Items for channel sounder

  - Control personal computer (Windows operation system)
  - Caesium standard (Microsemi 5071A)
  - Coaxial one-to-sixteen divider (**NO alternative**)
  - Signal generator for the Tx (R&S SMF100)
  - Signal generator for the Rx (Keysight E8267D)
  - Signal analyzer (R&S FSV7)
  - Frequency converter for the Tx with horn antenna and tripod adaptor (VDI WR3.4SGX-M)
  - Frequency converter for the Rx with horn antenna and tripod adaptor (VDI WR3.4CCD-M12)
  - 2 Tripods
  - 1 Coaxial cable (12 m, N-male to SMA-male)
  - 2 Coaxial cables (3 m, SMA)
  - 1 Coaxial cable (20 m, SMA)
  - 1 Coaxial cable (5 m, 2.92 mm)
  - 2 Coaxial cables (5 m, 3.5 mm)
  - 1 Ethernet cable

- Items for motion capture system
  - Control personal computer (Linux operation system)
  - Calibration square
  - 4 Servant personal computers (Linux operation system)
  - 5 Tripods
  - 4 Microsoft Azure Kinects
  - 4 USB cables (type-C, 1 m, **NO alternative**)
  - Ethernet hub (5 or more slots)

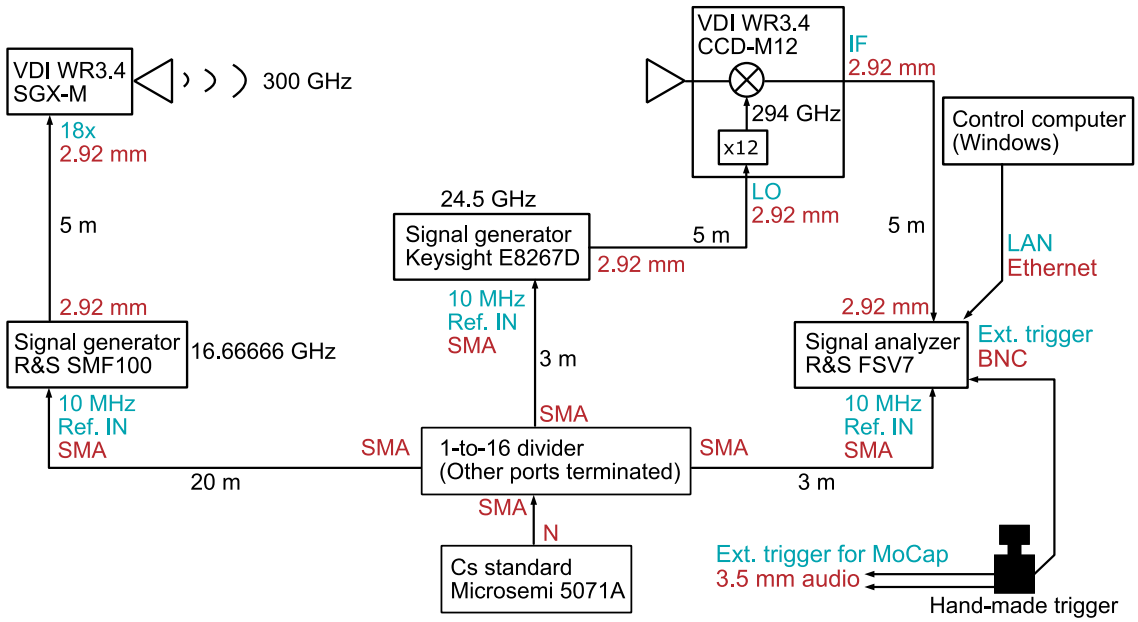

Figure A.1: The detailed block diagram of installation of the channel sounder at 300 GHz. The port names are identified by green font and the connector types are identified by red font.

- 2 Ethernet cables (10 m)
- 3 Ethernet cables (5 m)
- 2 Hand-made synchronization cables (10 m, 3.5 mm audio)

A.2.1 Dynamic Channel Sounder at 300 GHz

The detailed block diagram of installation is shown in Fig. A.1. The steps for installing channel sounder are described in the follows,

1. Deploy the devices.
2. Supply power to the Caesium standard as soon as possible.
3. Connect the instrument with cables.
4. Set up the instruments.
5. Test measurement.

The detailed processes are described as follows.

Firstly, deploy the instruments at the Tx and the Rx sites, respectively. For simple cabling, the Caesium standard (Microsemi 5071A) and the frequency converter (VDI

WR3.4CCD-M12), the signal generator (Keysight E8267D), and the signal analyzer (R&S FSV7) for the Rx should be deployed nearby. Likewise, the devices, i.e., the frequency converter (VDI WR3.4SGX-M) and the signal generator (R&S SMF100) for the Tx are located together.

Once the locations of the instruments are fixed, the Caesium standard should be powered as soon as possible for warming up (about 10 minutes). The Caesium standard will be booted automaticly. Check if the output frequency is 10 MHz before connecting the cables.

Then connect the cables between the instruments before booting. From the output port of the Caesium standard, the coaxial cable (12 m, N-male to SMA-male) is connected to the one-to-sixteen divider (S-port). Since the one-to-sixteen divider also acts as an attenuator, which prevents instruments from destruction from overload power, please **NEVER change the configuration of divider**, e.g., replacing with an one-to-four divider or removing the termination caps. If you have to change the configuration, make sure the 10 MHz reference signal is attenuated by a factor of 15 dB. The output ports of the divider are connected to the **ref. 10 MHz** ports of the signal generators and signal analyzer. The 20 m SMA cable is for the connection to the Tx site and two 3 m SMA cables are for the connection to the Rx site. In the Rx site, the frequency converter is connected to the output port of the signal generator from its local oscillator (LO) port (5 m, 2.92 mm connector) and is connected to the input port of the signal analyzer from its intermediate frequency (IF) port (5 m, 3.5 mm connector). The signal analyzer is connected to the control personal computer via the Ethernet port. Connect the BNC plug of the hand-made trigger cable to the external trigger port of the signal analyzer. In the Tx site, the input port (**x18**) of the frequency converter is connected to the output port of the signal generator (5 m, 3.5 mm connector).

After the connection, the instruments should be set up. First of all, connect the power cable to all the remaining instruments. From the **Utility** interface, the Caesium standard is reset. The signal generator at the Tx site is set by the frequency at 16.66666 GHz, the output power is 6 dBm, and the external reference source. The signal generator at the Rx site is set by the frequency at 24.5 GHz, the output power is 12 dBm, and the external reference source. If there is any change of the cable configuration, make sure the input powers are 0 dBm $\pm$3 dB and 0-6 dBm in the Tx and the Rx sides, respectively. Once the setting is confirmed correct, boot the frequency converter in the Tx site and turn on the output of the signal generators. The 300 GHz CW is transmitted and the down-converted signal is received by the signal analyzer.

Finally, the test measurement can be conducted. Since the sounding waveform is the CW, there is no setting for the Tx. The signal analyzer should be set up as follows,

1. Set the reference signal using the external source.

2. Set the center frequency to $300 - 294 = 6$ GHz.

3. Set the analyzer to single measurement mode.
4. Change the measurement mode to **IQAnalyzer**.
5. Set the measurement parameters as follows.
    - Sample rate: 300 kHz (in this thesis)
    - Trigger mode: "EXT" stands for external trigger.
    - Trigger slope: "POS" stand for starting the measurement when detecting the a positive slope of the rectangular trigger signal.
    - Measurement period: 20 seconds (in this thesis)
6. Set the threshold for the external trigger signal at 2 V.

The processes above can be automatically conducted by my C++ program "DynamicSoundingFSV/DynamicSounding.sln" via Microsoft Visual Studio. The parameters can be found and modified in "DynamicSoundingFSV/Parameters.json". The dependency, **VISA library provided by R&S**, must be installed in the control personal computer in advance. If the user have any difficulty about the C++ environment, MATLAB codes can be used for automatic setting, too. Be adviced that **there is a preparation time from sending commands until the analyzer is ready for trigger**. Once the analyzer is ready, press the trigger for starting the measurement. After the measurement, save the received complex **voltage** via USB memory or Ethernet (my program collects the results automatically).

A.2.2 Motion Capture using Microsoft Azure Kinects

The detailed block diagram of the installation is shown in Fig. A.2. The steps for installing the motion capture system are described as follows,

1. Install and deploy1 the camera units.
2. Connect the servant and control computers to the same local area network (LAN).
3. Connect the trigger cables for the cameras.
4. Calibration measurement.

The detailed processes are described as follows. Firstly, as shown in Fig. A.3, implement the camera units consisting of a tripod, a servant computer, a Microsoft Azure Kinect, and a 1 m USB type-C cable. **Never consider a sorter USB type-C cable** since the servant will fail to recognize the camera in any mode superior to USB 3.0. After the implementation of the

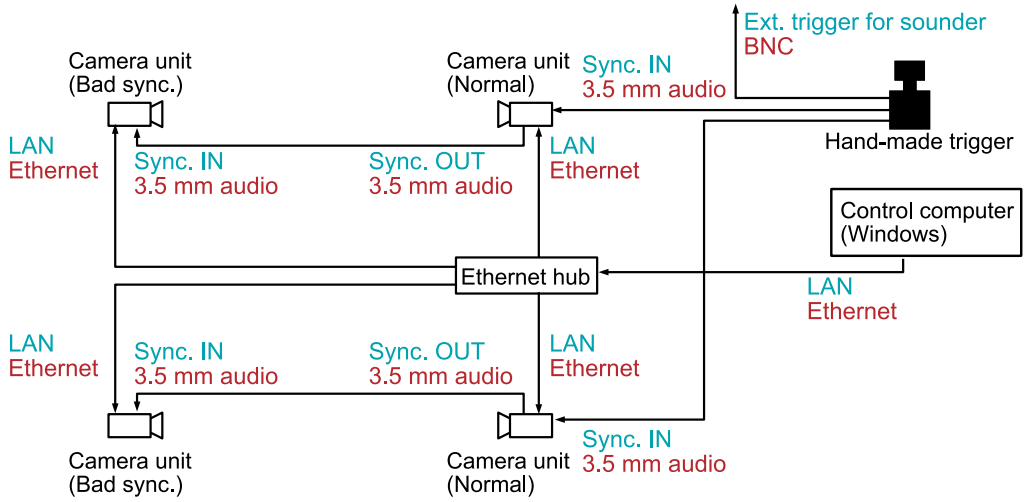

Figure A.2: The detailed block diagram of installation of the motion capture system. The port names are identified by green font and the connector types are identified by red font.

cameras units, deploy them to four corners of the measurement field. Be adviced the metalic tripods are good scatterer for radio waves when choosing the location for camera units. Furthermore, two of the Kinects with ill-functioning output synchronization ports, which is marked with **"Bad Sync."** must locate away from the control computer. Then connect all computers via the Ethernet hub for remote control. You can confirm the connection by "ping" command of Terminal. The two 3.5 mm plugs of the hand-made trigger should be connected to the normal Kinects and the hand-made 3.5 mm audio cables should be plugged into the **OUT port of the normal Kinects** and the **IN port of the ill-functioning Kinects.** After all the connection completed, the calibration measurement can be conducted. Deploy the callibration square in the center of the measurement field and make sure the square can be flimed by all the cameras. Via the remote control, start the measurements of all the cameras in the **subordinate mode**. The cameras will not start the measurement until the trigger signal is sent. Finally, collect the calibration videos containing the calibration square from four different viewpoints via the LAN.

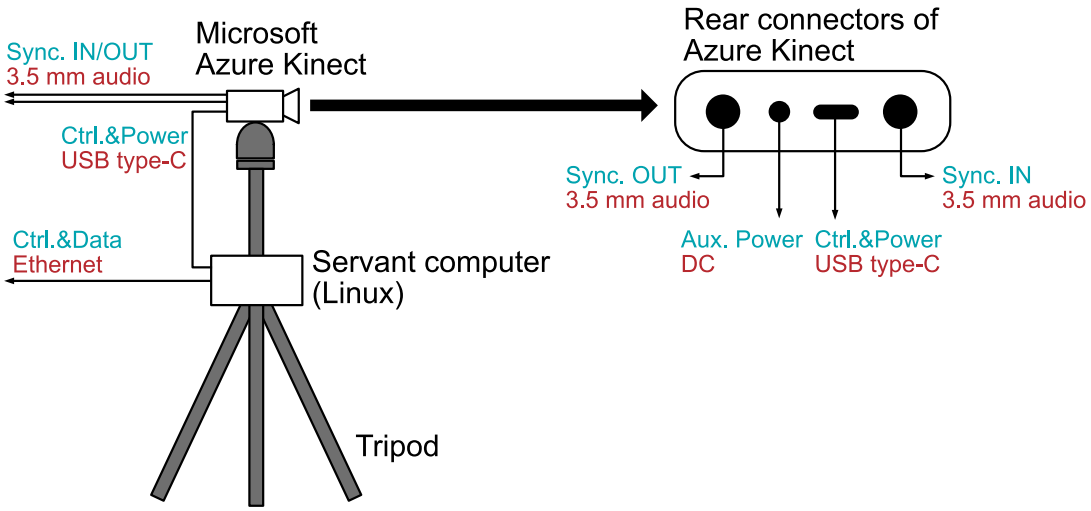

Figure A.3: The detailed block diagram of installation of each camera unit. The port names are identified by green font and the connector types are identified by red font.

A.3 Summary

The installation guide of the channel sounder and MoCap system have been presented in this appendix. By conducting the measurement step for both system and trigger at once, the synchronized human motion and its corresponding dynamic channel can be obtained.

Appendix B

Post-processing of the human motion

B.1 Introduction

In this chapter, the post-processing for the human motion mentioned in Chapter 4 and 5 is introduced. In Section B.2, the flow of the post-processing is presented. In Section B.3, the codes are desplayed.

B.2 Flow of the post-processing

In this section, the flow of the post-processing, that converts the 3D videos of human motion to the point cloud-represented human bodies, is introduced. The inputs are as follows,

- Calibration videos from four viewpoints
- Videos of the human motion from four viewpoints

Be advised that the configuration of the cameras must be the same for all the sets of videos. A update of the calibration videos is necessary after any tiny movement of the camera unit due to an accident.

The general flow chart is shown in Fig B.1. Firstly, the transformation matrices for combining four point clouds are generated. Since the calibration videos are static, the raging error of the depth cameras can be eased by averaging the depth of each pixel in multiple frame. The pixels of the IR light reflectors of the calibration square are then separated by a IR brightness filter. After converted to point clouds, the bright points are grouped into three clusters. Considering the point in each cluster closest to the camera should be the brightest one, the centroid of each reflector is calculated by adding the radius of the reflector in depth to the brightest point. The order of the centroids can be found by the side lengths, 9 cm, 12 cm, and 15 cm. Finally, the transformation matrices are defined to let the triangle from each view point fit to the calibration square located at the origin.

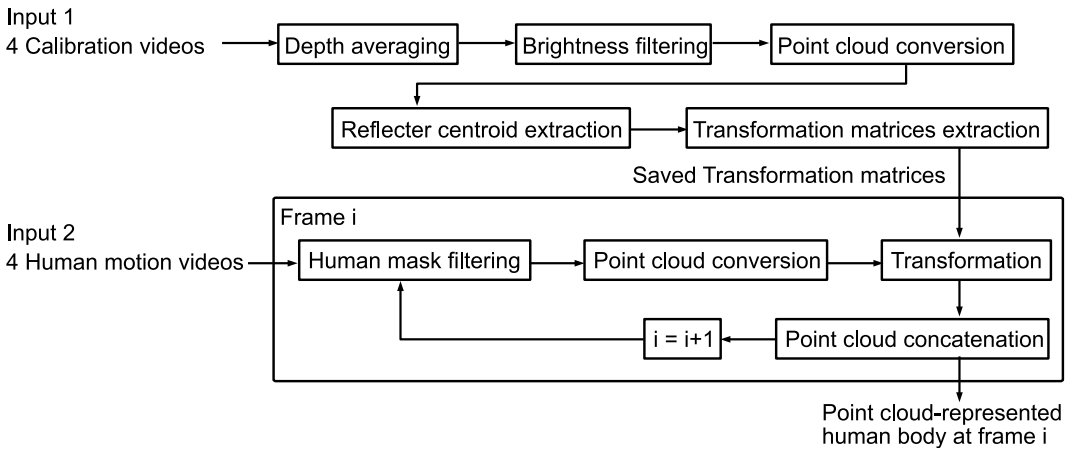

Figure B.1: The flow chart of the post-processing for transforming the videos of the human motion to point-cloud represented human bodies.

After the transformation matrices extraction, the point cloud-represented human bodies can by generated. To ease the noise points along the shadow boundary, the human mask identified by a machine-learning model [53] is used to filter the pixels of the human body. Finally, the depth map of the human body is converted to point cloud, transformed to the global coordination system, and conbined to the complete point cloud of the human body.

B.3 The souce codes of post-processing

B.3.1 Codes for transformation matrices extraction

The directory tree is as follows,

```
—SquareCal
  |--build
  | |--calib_k4a (Main program)
  | |--Laser_transform0.txt (Output transformation matrices)
  | |--Laser_transform1.txt
  | |--Laser_transform2.txt
  | |--Laser_transform3.txt
  |--include (Header files)
  | |--camera_extrinsics.hpp
  | |--LaserCalib.hpp
  |--input
  | |--yyyymmdd (Measurement date for identification)
  |--output (Calibrated point cloud for quick check)
  |--src (Source codes)
```

```
  | |--camera_extrinsics.cpp
  | |--LaserCalib.cpp
  |--CMakelists.txt
```

The source codes are desplayed below. The dependencies can be identified from **CMakelists.txt**.

Listing B.1: CMakelists.txt

```
cmake_minimum_required(VERSION 3.15)
project(calib_k4a LANGUAGES CXX)
set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
find_package(k4a REQUIRED)
find_package(k4abt REQUIRED)
find_package(k4arecord REQUIRED)
find_package(PCL 1.12 REQUIRED)
# Source
set(INCLUDE_FILES
 include/camera_extrinsics.hpp
 include/LaserCalib.hpp
)
set(SOURCE_FILES
 ${INCLUDE_FILES}
 src/camera_extrinsics.cpp
 src/LaserCalib.cpp
)
include_directories(${PCL_INCLUDE_DIRS})
link_directories(${PCL_LIBRARY_DIRS})
add_definitions(${PCL_DEFINITIONS})
# Targets
add_executable(calib_k4a ${SOURCE_FILES})
target_include_directories(calib_k4a PUBLIC include)
target_link_libraries(calib_k4a PUBLIC
 k4a # Kinect SDK
 k4abt
 k4arecord
 ${PCL_LIBRARIES}
 stdc++fs # filesystem
```

```
)
```

Listing B.2: include/LaserCalib.hpp

```
#pragma once
#include <fstream>
#include <filesystem>
#include "camera_extrinsics.hpp"
#include <k4arecord/playback.hpp>
#include <pcl/registration/
    transformation_estimation_dual_quaternion.h>
// Customized structure
typedef struct
{
 std::string filename;
 k4a_playback_t handle;
 k4a::playback playback;
 k4a_record_configuration_t record_config;
 k4a::calibration k4a_calibration;
 k4a::transformation k4a_transform;
 Eigen::Quaternionf rotation2master;
 Eigen::Matrix4f transform2master;
 k4a::capture capture;
 k4abt_body_t body;
 int idx_closest_body;
 std::vector<bool> conf_lev;
 pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr laserCalCloud;
 std::vector<Eigen::Vector3f> frame_cloud;
 int camera_id;
} recording_t;
// Declearation of the functions
void print_calibration(k4a::calibration& calibration);
uint64_t first_capture_timestamp(k4a_capture_t capture);
uint64_t capture_depth_timestamp(k4a::capture capture);
FrameInfo* process_capture_incbody(recording_t *file);
FrameInfo* process_capture(recording_t *file);
pcl::PointXYZRGBNormal Laser_coord_exract(recording_t *file, int
    reptime, std::chrono::duration<int, std::micro> laser_time, int
     color);
```

```
pcl::PointXYZRGBNormal IRSphere_coord_exract(recording_t *file,
   int reptime, std::chrono::duration<int, std::micro> IR_time);
void CopyIntrinsics(const k4a_calibration_camera_t& from,
   CameraIntrinsics& to);
void print_body_information(k4abt_body_t body);
// Definition of the functions
void print_calibration(k4a::calibration& calibration)
 {
  {
   std::cout << "Depth camera:" << std::endl;
   auto calib = calibration.depth_camera_calibration;
   std::cout << "resolution width:" << calib.resolution_width <<
      std::endl;
   std::cout << "resolution height:" << calib.resolution_height
     << std::endl;
   std::cout << "principal point x:" << calib.intrinsics.
      parameters.param.cx << std::endl;
   std::cout << "principal point y:" << calib.intrinsics.
      parameters.param.cy << std::endl;
   std::cout << "focal length x:" << calib.intrinsics.parameters.
      param.fx << std::endl;
   std::cout << "focal length y:" << calib.intrinsics.parameters.
      param.fy << std::endl;
   std::cout << "radial distortion coefficients:" << std::endl;
   std::cout << "k1:" << calib.intrinsics.parameters.param.k1 <<
      std::endl;
   std::cout << "k2:" << calib.intrinsics.parameters.param.k2 <<
      std::endl;
   std::cout << "k3:" << calib.intrinsics.parameters.param.k3 <<
      std::endl;
   std::cout << "k4:" << calib.intrinsics.parameters.param.k4 <<
      std::endl;
   std::cout << "k5:" << calib.intrinsics.parameters.param.k5 <<
      std::endl;
   std::cout << "k6:" << calib.intrinsics.parameters.param.k6 <<
      std::endl;
   std::cout << "center of distortion in Z=1 plane, x:" << calib.
      intrinsics.parameters.param.codx << std::endl;
```

```
  std::cout << "center-of-distortion-in-Z=1-plane,-y:-" << calib.
     intrinsics.parameters.param.cody << std::endl;
  std::cout << "tangential-distortion-coefficient-x:-" << calib.
     intrinsics.parameters.param.p1 << std::endl;
  std::cout << "tangential-distortion-coefficient-y:-" << calib.
     intrinsics.parameters.param.p2 << std::endl;
  std::cout << "metric-radius:-" << calib.intrinsics.parameters.
     param.metric_radius << std::endl;
 }
 {
  std::cout << "Color-camera:" << std::endl;
  auto calib = calibration.color_camera_calibration;
  std::cout << "resolution-width:-" << calib.resolution_width <<
     std::endl;
  std::cout << "resolution-height:-" << calib.resolution_height
     << std::endl;
  std::cout << "principal-point-x:-" << calib.intrinsics.
     parameters.param.cx << std::endl;
  std::cout << "principal-point-y:-" << calib.intrinsics.
     parameters.param.cy << std::endl;
  std::cout << "focal-length-x:-" << calib.intrinsics.parameters.
     param.fx << std::endl;
  std::cout << "focal-length-y:-" << calib.intrinsics.parameters.
     param.fy << std::endl;
  std::cout << "radial-distortion-coefficients:" << std::endl;
  std::cout << "k1:-" << calib.intrinsics.parameters.param.k1 <<
     std::endl;
  std::cout << "k2:-" << calib.intrinsics.parameters.param.k2 <<
     std::endl;
  std::cout << "k3:-" << calib.intrinsics.parameters.param.k3 <<
     std::endl;
  std::cout << "k4:-" << calib.intrinsics.parameters.param.k4 <<
     std::endl;
  std::cout << "k5:-" << calib.intrinsics.parameters.param.k5 <<
     std::endl;
  std::cout << "k6:-" << calib.intrinsics.parameters.param.k6 <<
     std::endl;
  std::cout << "center-of-distortion-in-Z=1-plane,-x:-" << calib.
```

```
      intrinsics.parameters.param.codx << std::endl;
    std::cout << "center of distortion in Z=1 plane, y: " << calib.
      intrinsics.parameters.param.cody << std::endl;
    std::cout << "tangential distortion coefficient x: " << calib.
      intrinsics.parameters.param.p1 << std::endl;
    std::cout << "tangential distortion coefficient y: " << calib.
      intrinsics.parameters.param.p2 << std::endl;
    std::cout << "metric radius: " << calib.intrinsics.parameters.
      param.metric_radius << std::endl;
  }
  auto extrinsics = calibration.extrinsics[
    K4A_CALIBRATION_TYPE_DEPTH][K4A_CALIBRATION_TYPE_COLOR];
  std::cout << "depth2color translation: (" << extrinsics.
    translation[0] << "," << extrinsics.translation[1] << "," <<
    extrinsics.translation[2] << ")" << std::endl;
  std::cout << "depth2color rotation: |" << extrinsics.rotation[0]
     << "," << extrinsics.rotation[1] << "," << extrinsics.
    rotation[2] << "|" << std::endl;
  std::cout << "          |" << extrinsics.rotation[3] << "," <<
    extrinsics.rotation[4] << "," << extrinsics.rotation[5] << "|
    " << std::endl;
  std::cout << "          |" << extrinsics.rotation[6] << "," <<
    extrinsics.rotation[7] << "," << extrinsics.rotation[8] << "|
    " << std::endl;
 }
uint64_t first_capture_timestamp(k4a_capture_t capture)
{
 uint64_t min_timestamp = (uint64_t)-1;
 k4a_image_t images[3];
 images[0] = k4a_capture_get_color_image(capture);
 images[1] = k4a_capture_get_depth_image(capture);
 images[2] = k4a_capture_get_ir_image(capture);
 for (int i = 0; i < 3; i++)
 {
  if (images[i] != NULL)
  {
   uint64_t timestamp = k4a_image_get_device_timestamp_usec(images
     [i]);
```

```
    if (timestamp < min_timestamp)
    {
     min_timestamp = timestamp;
    }
    k4a_image_release(images[i]);
    images[i] = NULL;
   }
  }
  return min_timestamp;
}
uint64_t capture_depth_timestamp(k4a::capture capture)
{
 k4a::image image;
 uint64_t timestamp = 0;
 image = capture.get_depth_image();
 if (image != NULL)
 {
  uint64_t timestamp = image.get_device_timestamp().count();
  image.reset();
 }
 return timestamp;
}
FrameInfo* process_capture_incbody(recording_t *file)
{
 k4a::image images[3];
 FrameInfo* frame = new FrameInfo();
 k4abt::tracker tracker = k4abt::tracker::create(file->
    k4a_calibration);
 if (!tracker.enqueue_capture(file->capture))
 {
  // It should never hit timeout when K4A_WAIT_INFINITE is set.
  std::cout << "Error! Add capture to tracker process queue
     timeout!" << std::endl;
 }
 k4abt::frame body_frame = tracker.pop_result();
 tracker.shutdown();
 if (body_frame != nullptr)
 {
```

```
  uint32_t num_bodies = body_frame.get_num_bodies();
  if(num_bodies)
  {
   int idx_closest_body = 0;
   for(int idx_body = 0; idx_body < num_bodies; idx_body++) {
    k4abt_body_t closest_body = body_frame.get_body(
       idx_closest_body);
    k4abt_body_t current_body = body_frame.get_body(idx_body);
    if(closest_body.skeleton.joints[0].position.v[2] >
       current_body.skeleton.joints[0].position.v[2])
       idx_closest_body = idx_body;
    std::cout << closest_body.skeleton.joints[0].position.v[2] <<"
       ,"<< current_body.skeleton.joints[0].position.v[2] << std::
       endl;
   }
   std::cout << num_bodies <<","<< idx_closest_body << std::endl;
   k4abt_body_t body = body_frame.get_body(idx_closest_body);
   images[2] = body_frame.get_body_index_map();
   frame->idx_closest_body = idx_closest_body;
  }
 }
 images[0] = file->capture.get_color_image();
 images[1] = file->capture.get_depth_image();
 // Copy depth
 frame->DepthWidth = images[1].get_width_pixels();
 frame->DepthHeight = images[1].get_height_pixels();
 frame->DepthStride = images[1].get_stride_bytes();
 const unsigned depth_size = frame->DepthStride * frame->
    DepthHeight;
 frame->DepthImage.resize(depth_size);
 memcpy(frame->DepthImage.data(), reinterpret_cast<const uint16_t
    *>( images[1].get_buffer()), depth_size * 2);
 // Copy body map
 frame->BodyMapWidth = images[2].get_width_pixels();
 frame->BodyMapHeight = images[2].get_height_pixels();
 frame->BodyMapStride = images[2].get_stride_bytes();
 const unsigned bodymap_size = frame->BodyMapStride * frame->
    BodyMapHeight;
```

```
frame->BodyMapImage.resize(bodymap_size);
memcpy(frame->BodyMapImage.data(), reinterpret_cast<const uint8_t
    *>( images[2].get_buffer()), bodymap_size);
k4a::image transformed_color_image = file->k4a_transform.
    color_image_to_depth_camera(images[1], images[0]);
k4a::image point_cloud_image = file->k4a_transform.
    depth_image_to_point_cloud(images[1],
    K4A_CALIBRATION_TYPE_DEPTH);
// Copy point cloud
const size_t cloud_buf_size = point_cloud_image.get_size();
frame->PointCloudData.resize(cloud_buf_size * 2);
memcpy(frame->PointCloudData.data(), reinterpret_cast<const
    int16_t*>( point_cloud_image.get_buffer() ), cloud_buf_size );
// Copy color
frame->ColorWidth = transformed_color_image.get_width_pixels();
frame->ColorHeight = transformed_color_image.get_height_pixels();
frame->ColorStride = transformed_color_image.get_stride_bytes();
const uint8_t* color_image = \
 reinterpret_cast<const uint8_t*>( transformed_color_image.
     get_buffer());
const size_t color_size = transformed_color_image.get_size();
frame->ColorImage.resize(color_size);
memcpy(frame->ColorImage.data(), color_image, color_size);
// Copy the transformation matrix (to master camera)
frame->transform2master = Eigen::Matrix4f(file->transform2master)
    ;
// Resource recycle
transformed_color_image.reset();
point_cloud_image.reset();
printf("%-32s", file->filename.c_str());
for (int i = 0; i < 2; i++)
{
 if (images[i] != NULL)
 {
  frame->timestamp = images[i].get_device_timestamp().count();
  printf("--%7ju-usec", frame->timestamp);
  images[i].reset();
  images[i] = NULL;
```

```
    }
    else
    {
      printf("--%12s", "");
    }
  }
  printf("\n");
  return frame;
}
FrameInfo* process_capture(recording_t *file)
{
  k4a::image images[3];
  FrameInfo* frame = new FrameInfo();
  images[0] = file->capture.get_color_image();
  images[1] = file->capture.get_depth_image();
  images[2] = file->capture.get_ir_image();
  // Copy depth
  frame->DepthWidth = images[1].get_width_pixels();
  frame->DepthHeight = images[1].get_height_pixels();
  frame->DepthStride = images[1].get_stride_bytes();
  const unsigned depth_size = frame->DepthStride * frame->
      DepthHeight;
  frame->DepthImage.resize(depth_size);
  memcpy(frame->DepthImage.data(), reinterpret_cast<const uint16_t
      *>( images[1].get_buffer()), depth_size * 2);
  k4a::image transformed_color_image = file->k4a_transform.
      color_image_to_depth_camera(images[1], images[0]);
  k4a::image point_cloud_image = file->k4a_transform.
      depth_image_to_point_cloud(images[1],
      K4A_CALIBRATION_TYPE_DEPTH);
  // Copy point cloud
  const size_t cloud_size = point_cloud_image.get_size();
  frame->PointCloudData.reserve(cloud_size * 2);
  frame->PointCloudData.resize(cloud_size);
  memcpy(frame->PointCloudData.data(), reinterpret_cast<const
      int16_t*>(point_cloud_image.get_buffer() ), cloud_size );
  // Copy color
  frame->ColorWidth = transformed_color_image.get_width_pixels();
```

```
frame->ColorHeight = transformed_color_image.get_height_pixels();
frame->ColorStride = transformed_color_image.get_stride_bytes();
const uint8_t* color_image = \
 reinterpret_cast<const uint8_t*>( transformed_color_image.
    get_buffer());
const size_t color_size = transformed_color_image.get_size();
frame->ColorImage.resize(color_size);
memcpy(frame->ColorImage.data(), color_image, color_size);
// Copy IR
frame->IRWidth = images[2].get_width_pixels();
frame->IRHeight = images[2].get_height_pixels();
frame->IRStride = images[2].get_stride_bytes();
frame->IRImage.resize(images[2].get_size());
memcpy(frame->IRImage.data(), reinterpret_cast<const uint16_t*>(
   images[2].get_buffer()), images[2].get_size());
// Copy the transformation matrix (to master camera)
frame->transform2master = Eigen::Matrix4f(file->transform2master)
   ;
// Resource recycle
transformed_color_image.reset();
point_cloud_image.reset();
// printf("%-32s", file->filename.c_str());
for (int i = 0; i < 2; i++)
{
 if (images[i] != NULL)
 {
  frame->timestamp = images[i].get_device_timestamp().count() - (
     file->record_config.start_timestamp_offset_usec);
  // printf("  %7ju usec", frame->timestamp);
  images[i].reset();
  images[i] = NULL;
 }
 else
 {
  printf("--%12s", "");
 }
}
// printf("\n");
```

```
 return frame;
}
pcl::PointXYZRGBNormal Laser_coord_exract(recording_t *file, const
    int reptime, std::chrono::duration<int, std::micro> laser_time
   , int color){
 Eigen::Vector3f LaserCenter{0, 0, 0};
 int frame_rate = 30;
 switch(file->record_config.camera_fps){
  case K4A_FRAMES_PER_SECOND_5 :
   frame_rate = 5;break;
  case K4A_FRAMES_PER_SECOND_15 :
   frame_rate = 15;break;
  case K4A_FRAMES_PER_SECOND_30 :
   frame_rate = 30;break;
  default :
   printf("ERROR: Failed to fine frame rate setting!\n");
 }
  // Find the nearest frame to the assigned time stamp.
 std::cout << laser_time.count() << std::endl;
 file->playback.seek_timestamp(laser_time,
    K4A_PLAYBACK_SEEK_DEVICE_TIME);
 if (!file->playback.get_next_capture(&file->capture))
 {
  printf("ERROR: Recording file is empty: %s\n", file->filename.
     c_str());
 }
 // Transform depth image to cloud
 FrameInfo* frame = process_capture(file);
 std::cout << frame->timestamp << std::endl;
 if (frame->timestamp - laser_time.count() > 500000/ frame_rate)
    file->playback.get_previous_capture(&file->capture);
 pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr source_cloud (new
    pcl::PointCloud<pcl::PointXYZRGBNormal>);
 pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr cloud (new pcl::
    PointCloud<pcl::PointXYZRGBNormal>);
 pcl::Indices indices;
 const int count = frame->ColorWidth * frame->ColorHeight;
 const int coord_stride = 1;
```

```
const int rep_count = 20;
source_cloud->reserve(count);
std::vector<int> CloudBuf(count * 3);
std::vector<int> ColorBuf(count * 3);
std::vector<int> valid(count * 3, 0);
std::cout <<"Generating Laser Cloud From Frames, count ="<<
   count << std::endl;
for(int rep_idx = 0; rep_idx < reptime; rep_idx++){
 frame = process_capture(file);
 for (int i_px = 0; i_px < count * 3; i_px++) {
  int PCData_cast = (int) frame->PointCloudData.data()[i_px];
  CloudBuf[i_px] += PCData_cast;
  valid[i_px] += (int) (PCData_cast != 0);
  if((i_px-2) % 3 == 0 && (PCData_cast != 0)){
   int RData_cast = frame->ColorImage.data()[4 * (i_px-2) / 3 +
      2];
   int GData_cast = frame->ColorImage.data()[4 * (i_px-2) / 3 +
      1];
   int BData_cast = frame->ColorImage.data()[4 * (i_px-2) / 3 +
      0];
   ColorBuf[(i_px)] = RData_cast;
   ColorBuf[(i_px + 1)] = GData_cast;
   ColorBuf[(i_px + 2)] = BData_cast;
  }
 }
 file->playback.get_next_capture(&file->capture);
}
for (int i_pt = 0; i_pt < count; i_pt += coord_stride) {
 if (valid[3 * i_pt + 2] == 0) continue;
 if (CloudBuf.data()[3 * i_pt + 2] == 0) continue;
 std::vector<uint8_t> rgbPt{(uint8_t)frame->ColorImage.data()[4 *
     i_pt + 2], (uint8_t)frame->ColorImage.data()[4 * i_pt + 1],
    (uint8_t)frame->ColorImage.data()[4 * i_pt + 0]};
 std::vector<uint8_t>::iterator id_cmax = std::max_element(rgbPt.
    begin(), rgbPt.end());
 std::vector<uint8_t>::iterator id_cmin = std::min_element(rgbPt.
    begin(), rgbPt.end());
 size_t id_max = std::distance(rgbPt.begin(), id_cmax);
```

```
if (rgbPt[0] == rgbPt[1] && rgbPt[1] == rgbPt[2]) continue;
int16_t H = 0;
switch(id_max){
 case 0:
  H = 60 * ((int16_t)rgbPt[1] - (int16_t)rgbPt[2])/(int16_t)((*
     id_cmax)-(*id_cmin));
  break;
 case 1:
  H = 60 * ((int16_t)rgbPt[2] - (int16_t)rgbPt[0])/(int16_t)((*
     id_cmax)-(*id_cmin));
  break;
 case 2:
  H = 60 * ((int16_t)rgbPt[0] - (int16_t)rgbPt[1])/(int16_t)((*
     id_cmax)-(*id_cmin));
  break;
}
if (id_max != color) continue;
if (color == 0 && (H < -30 && H > 10)) continue; // filter the
   color in interest
if (color == 1 && (H < -40 && H > 20)) continue; // filter the
   color in interest
if (color == 2 && (H < -30 && H > 10)) continue; // filter the
   color in interest
if((*id_cmax) < (char) (100)) continue; // Find the points whose
    brightness (Value) > V/255, valid only when the exposure of
   the camera was squeezed.
if((*id_cmin) > (char) (0.5 * (*id_cmax)) ) continue; // Find
   the point red enough (S > 50% => (100-S)%)
// BGR -> RGB
pcl::PointXYZRGBNormal point;
point.x = CloudBuf.data()[3 * i_pt + 0] / 1000.0f / (float)valid
   [3 * i_pt + 0];
point.y = CloudBuf.data()[3 * i_pt + 1] / 1000.0f / (float)valid
   [3 * i_pt + 1];
point.z = CloudBuf.data()[3 * i_pt + 2] / 1000.0f / (float)valid
   [3 * i_pt + 2];
point.r = (uint8_t)frame->ColorImage.data()[4 * i_pt + 2];
point.g = (uint8_t)frame->ColorImage.data()[4 * i_pt + 1];
```

```
  point.b = (uint8_t)frame->ColorImage.data()[4 * i_pt + 0];
  if(std::isinf(point.x)||std::isinf(point.y)||std::isinf(point.z)
     ) continue;
  if (point.r == 0 && point.g == 0 && point.b == 0) continue;
  source_cloud->points.push_back(point);
}
std::cout << color << "-Color-filtered" << std::endl;
pcl::StatisticalOutlierRemoval<pcl::PointXYZRGBNormal> sor;
sor.setInputCloud (source_cloud);
sor.setMeanK (40);
sor.setStddevMulThresh (1.0);
sor.filter (*cloud);
pcl::removeNaNFromPointCloud(*cloud, *cloud, indices);
std::cout <<"Constructed-PCL-PointCloud-for-Laser-Marker:-" <<std
    ::endl;
pcl::NormalEstimationOMP<pcl::PointXYZRGBNormal, pcl::
    PointXYZRGBNormal> ne;
const double normal_radius = 0.01;
ne.setInputCloud (cloud);
// Create an empty kdtree representation, and pass it to the
    normal estimation object.
// Its content will be filled inside the object, based on the
    given input dataset (as no other search surface is given).
pcl::search::Octree<pcl::PointXYZRGBNormal>::Ptr tree (new pcl::
    search::Octree<pcl::PointXYZRGBNormal>(normal_radius));
ne.setSearchMethod (tree);
// Use all neighbors in a sphere of radius 3cm
ne.setRadiusSearch (normal_radius);
ne.useSensorOriginAsViewPoint();
ne.compute (*cloud);
// Resource recycling;
file->capture.reset();
for(auto pt: cloud->points){
 LaserCenter(0) += pt.x;
 LaserCenter(1) += pt.y;
 LaserCenter(2) += pt.z;
}
LaserCenter = LaserCenter / (float)cloud->size();
```

```
 std::string filename = "../output/Cfilteredcloud_" + std::
    to_string(color) + "Cam" + std::to_string(file->camera_id) + "
    .pcd";
 pcl::io::savePCDFileBinary(filename, *cloud);
 cloud->clear();
 return pcl::PointXYZRGBNormal(LaserCenter(0), LaserCenter(1),
    LaserCenter(2), (uint8_t)(255*(color == 0)), (uint8_t)(255*(
    color == 1)), (uint8_t)(255*(color == 2)));
}
inline pcl::PointXYZRGBNormal IRSphere_coord_exract(recording_t *
   file, int reptime, std::chrono::duration<int, std::micro>
   IR_time)
{
 Eigen::Vector3f LaserCenter{0, 0, 0};
 int frame_rate = 30;
 switch(file->record_config.camera_fps){
  case K4A_FRAMES_PER_SECOND_5 :
   frame_rate = 5;break;
  case K4A_FRAMES_PER_SECOND_15 :
   frame_rate = 15;break;
  case K4A_FRAMES_PER_SECOND_30 :
   frame_rate = 30;break;
  default :
   printf("ERROR: Failed to fine frame rate setting!\n");
 }
  // Find the nearest frame to the assigned time stamp.
 std::cout << IR_time.count() << std::endl;
 file->playback.seek_timestamp(IR_time,
    K4A_PLAYBACK_SEEK_DEVICE_TIME);
 if (!file->playback.get_next_capture(&file->capture))
 {
  printf("ERROR: Recording file is empty: %s\n", file->filename.
     c_str());
 }
 // Transform depth image to cloud
 FrameInfo* frame = process_capture(file);
 std::cout << frame->timestamp << std::endl;
 if (frame->timestamp - IR_time.count() > 500000/ frame_rate) file
```

```
    ->playback.get_previous_capture(&file->capture);
 pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr source_cloud (new
    pcl::PointCloud<pcl::PointXYZRGBNormal>);
 pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr cloud (new pcl::
    PointCloud<pcl::PointXYZRGBNormal>);
 pcl::Indices indices;
 const int count = frame->ColorWidth * frame->ColorHeight;
 const int coord_stride = 1;
 const int rep_count = 20;
 source_cloud->reserve(count);
 std::vector<int> CloudBuf(frame->PointCloudData.size());
 std::vector<int> IRBuf(frame->IRImage.size());
 std::vector<int> valid(count, 0);
 std::cout <<"Generating Cloud From Frames, count ="<< count <<
    std::endl;
 for(int rep_idx = 0; rep_idx < rep_count; rep_idx++){
  frame = process_capture(file);
  for (int i_px = 0; i_px < count; i_px++) {
   int PCData_cast = (int) frame->PointCloudData.data()[i_px*3 +
      2];
   CloudBuf[i_px] += PCData_cast;
   int IRData_cast = (int) frame->IRImage.data()[i_px];
   IRBuf[i_px] += IRData_cast;
   valid[i_px] += (int) (PCData_cast != 0);
  }
  file->playback.get_next_capture(&file->capture);
 }
 for (int i_pt = 0; i_pt < count; i_pt += coord_stride) {
  if (valid[3 * i_pt + 2] == 0) continue;
  if (CloudBuf.data()[3 * i_pt + 2] == 0) continue;
  if (IRBuf.data()[3 * i_pt + 2]/valid[i_pt] < 1000) continue;
  // BGR -> RGB
  pcl::PointXYZRGBNormal point;
  point.x = CloudBuf.data()[3 * i_pt + 0] / 1000.0f / (float)valid
     [i_pt];
  point.y = CloudBuf.data()[3 * i_pt + 1] / 1000.0f / (float)valid
     [i_pt];
  point.z = CloudBuf.data()[3 * i_pt + 2] / 1000.0f / (float)valid
```

```
    [i_pt];
  point.r = (uint8_t)frame->ColorImage.data()[4 * i_pt + 2];
  point.g = (uint8_t)frame->ColorImage.data()[4 * i_pt + 1];
  point.b = (uint8_t)frame->ColorImage.data()[4 * i_pt + 0];
  if(std::isinf(point.x)||std::isinf(point.y)||std::isinf(point.z)
    ) continue;
  if (point.r == 0 && point.g == 0 && point.b == 0) continue;
}
pcl::StatisticalOutlierRemoval<pcl::PointXYZRGBNormal> sor;
sor.setInputCloud (source_cloud);
sor.setMeanK (40);
sor.setStddevMulThresh (1.0);
sor.filter (*cloud);
pcl::removeNaNFromPointCloud(*cloud, *cloud, indices);
std::cout <<"Constructed PCL PointCloud for Laser Marker:" <<std
    ::endl;
pcl::NormalEstimationOMP<pcl::PointXYZRGBNormal, pcl::
    PointXYZRGBNormal> ne;
const double normal_radius = 0.01;
ne.setInputCloud (cloud);
// Create an empty kdtree representation, and pass it to the
    normal estimation object.
// Its content will be filled inside the object, based on the
    given input dataset (as no other search surface is given).
pcl::search::Octree<pcl::PointXYZRGBNormal>::Ptr tree (new pcl::
    search::Octree<pcl::PointXYZRGBNormal>(normal_radius));
ne.setSearchMethod (tree);
// Use all neighbors in a sphere of radius 3cm
ne.setRadiusSearch (normal_radius);
ne.useSensorOriginAsViewPoint();
ne.compute (*cloud);
// Resource recycling;
file->capture.reset();
for(auto pt: cloud->points){
  LaserCenter(0) += pt.x;
  LaserCenter(1) += pt.y;
  LaserCenter(2) += pt.z;
}
```

```
 LaserCenter = LaserCenter / (float)cloud->size();
 std::string filename = "../output/IRfilteredcloud_Cam" + std::
    to_string(file->camera_id) + ".pcd";
 pcl::io::savePCDFileBinary(filename, *cloud);
 cloud->clear();
 return pcl::PointXYZRGBNormal(LaserCenter(0), LaserCenter(1),
    LaserCenter(2), (uint8_t)(255), (uint8_t)(0), (uint8_t)(0));
}
void CopyIntrinsics(
 const k4a_calibration_camera_t& from,
 CameraIntrinsics& to)
{
 to.Width = from.resolution_width;
 to.Height = from.resolution_height;
 const k4a_calibration_intrinsic_parameters_t& params = from.
    intrinsics.parameters;
 to.cx = params.param.cx;
 to.cy = params.param.cy;
 to.fx = params.param.fx;
 to.fy = params.param.fy;
 to.k[0] = params.param.k1;
 to.k[1] = params.param.k2;
 to.k[2] = params.param.k3;
 to.k[3] = params.param.k4;
 to.k[4] = params.param.k5;
 to.k[5] = params.param.k6;
 to.codx = params.param.codx;
 to.cody = params.param.cody;
 to.p1 = params.param.p1;
 to.p2 = params.param.p2;
}
void CalibrationFromK4a(
 const k4a_calibration_t& from,
 CameraCalibration& to)
{
 CopyIntrinsics(from.depth_camera_calibration, to.Depth);
 CopyIntrinsics(from.color_camera_calibration, to.Color);
 // Extrinsics from depth to color camera
```

```
 const k4a_calibration_extrinsics_t* extrinsics = &from.extrinsics
     [K4A_CALIBRATION_TYPE_DEPTH][K4A_CALIBRATION_TYPE_COLOR];
 for (int i = 0; i < 9; ++i) {
  to.RotationFromDepth[i] = extrinsics->rotation[i];
 }
 for (int i = 0; i < 3; ++i) {
  to.TranslationFromDepth[i] = extrinsics->translation[i];
 }
}
void print_body_information(k4abt_body_t body)
{
 std::cout << "Body ID: " << body.id << std::endl;
 FILE* outcsv;
 outcsv = fopen("./body_frames.csv", "a");
 fprintf(outcsv, "New body\n");
 for (int i = 0; i < (int)K4ABT_JOINT_COUNT; i++)
 {
  k4a_float3_t position = body.skeleton.joints[i].position;
  k4a_quaternion_t orientation = body.skeleton.joints[i].
     orientation;
  k4abt_joint_confidence_level_t confidence_level = body.skeleton.
     joints[i].confidence_level;
  fprintf(outcsv, "Joint[%d]: Position[mm] ( %f, %f, %f ); 
     Orientation ( %f, %f, %f, %f); Confidence Level (%d) \n",
   i, position.v[0], position.v[1], position.v[2], orientation.v
      [0], orientation.v[1], orientation.v[2], orientation.v[3],
      confidence_level);
 }
 fprintf(outcsv, "\n\n");
 fclose(outcsv);
}
```

Listing B.3: include/camera_extrinsics.hpp

```
#pragma once
#include <cstdint>
#include <vector>
#include <iostream>
#include <Eigen/Eigen>
```

```
#include <k4a/k4a.hpp>
#include <k4arecord/playback.hpp>
#include <k4abt.hpp>
#include <pcl/ModelCoefficients.h>
#include <pcl/io/pcd_io.h>
#include <pcl/common/io.h>
#include <pcl/common/transforms.h>
#include <pcl/common/geometry.h>
#include <pcl/features/integral_image_normal.h>
#include <pcl/filters/passthrough.h>
#include <pcl/filters/extract_indices.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/filters/statistical_outlier_removal.h>
#include <pcl/search/octree.h>
#include <pcl/features/normal_3d_omp.h>
#include <pcl/features/principal_curvatures.h>
#include <pcl/registration/icp.h>
#include <pcl/sample_consensus/method_types.h>
#include <pcl/sample_consensus/model_types.h>
#include <pcl/segmentation/sac_segmentation.h>
#include <pcl/segmentation/extract_clusters.h>
#define BOXCLIP
// Customized structure
struct  CameraExtrinsics
{
 Eigen::Vector3f translation;
 Eigen::Quaternionf rotation;
};
struct CameraIntrinsics
{
 // Sensor resolution
 int32_t Width, Height;
 // Intrinsics
 float cx, cy;
 float fx, fy;
 float k[6];
 float codx, cody;
 float p1, p2;
```

```
};
struct CameraCalibration
{
 // Intrinsics for each camera
 CameraIntrinsics Color, Depth;
 // Extrinsics transform from 3D depth camera point to 3D point
    relative to color camera
 float RotationFromDepth[3*3];
 float TranslationFromDepth[3];
};
struct FrameInfo
{
 // Accelerometer reading for extrinsics calibration
 float Accelerometer[3];
 // Body including the joints, etc.
 Eigen::Matrix4f transform2master = Eigen::Matrix4f::Identity(4,
    4);
 Eigen::Quaternionf rotation2master;
 Eigen::Vector3f pelvis;
 // Color image
 std::vector<uint8_t> ColorImage;
 int ColorWidth = 0, ColorHeight = 0, ColorStride = 0;
 // Depth image
 std::vector<uint16_t> DepthImage;
 int DepthWidth = 0, DepthHeight = 0, DepthStride = 0;
 // IR image
 std::vector<uint16_t> IRImage;
 int IRWidth = 0, IRHeight = 0, IRStride = 0;
 // Body map image
 std::vector<uint8_t> BodyMapImage;
 int BodyMapWidth = 0, BodyMapHeight = 0, BodyMapStride = 0;
 int idx_closest_body = 0;
 // Point cloud data
 std::vector<int16_t> PointCloudData;
 uint32_t CameraIndex;
 int FrameNumber;
 std::string filename;
 uint64_t timestamp = 0;
```

```
};
struct AlignmentTransform
{
 float Transform[16];
 bool Identity = true;
 inline void operator=(const Eigen::Matrix4f& src)
 {
  Identity = src.isIdentity();
  for (int row = 0; row < 4; ++row) {
   for (int col = 0; col < 4; ++col) {
    Transform[row * 4 + col] = src(row, col);
   }
  }
 }
 inline void Set(Eigen::Matrix4f& dest) const
 {
  if (Identity) {
   dest = Eigen::Matrix4f::Identity();
  } else {
   for (int row = 0; row < 4; ++row) {
    for (int col = 0; col < 4; ++col) {
     dest(row, col) = Transform[row * 4 + col];
    }
   }
  }
 }
};
class ExtrinsicsCalibration
{
 public:
  ExtrinsicsCalibration()
  {
  }
  ~ExtrinsicsCalibration()
  {
  }
  bool CalculateExtrinsics(
   const std::vector<FrameInfo*>& frames,
```

```
    std::vector<AlignmentTransform>& extrinsics,
    std::vector<uint64_t>& timestamps);
 private:
  std::vector<pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr>
      full_cloud;
  bool GenerateFullCloudFromFrames(FrameInfo frame);
  bool GenerateBodyCloudFromFrames(FrameInfo frame);
  bool GenerateBGCloudFromFrames(FrameInfo frame);
  bool DownSample(pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr&
      cloud, const float voxel_size, const int idx);
};
```

Listing B.4: include/LaserCalib.cpp

```
#include <stdio.h>
#include <malloc.h>
#include <iostream>
#include "LaserCalib.hpp"
\\ #define ManualRefine
int main(int argc, char **argv)
{
 if (argc != 2)
 {
  printf("Usage: ./calib_k4a <date_yyyymmdd>\n");
  return 1;
 }
 // size_t file_count = (size_t)(argc - 2);
 size_t file_count = (size_t) 4;
 bool master_found = false;
 k4a_result_t result = K4A_RESULT_SUCCEEDED;
 k4a_stream_result_t stream_result = K4A_STREAM_RESULT_SUCCEEDED;
 int frame_rate = 30;
 std::string dir_Laser_input;
 dir_Laser_input = std::string("../../../THzExp/") + std::string(
     argv[1]) + std::string("/SquareCal/");
 // Allocate memory to store the state of N recordings.
 recording_t *files = reinterpret_cast<recording_t*>(malloc(sizeof
     (recording_t) * file_count));
 if (files == NULL)
```

```
{
 printf("Failed to allocate memory for playback (%zu bytes)\n",
    sizeof(recording_t) * file_count);
 return 1;
}
memset(files, 0, sizeof(recording_t) * file_count);
recording_t *cal_files = reinterpret_cast<recording_t*>(malloc(
   sizeof(recording_t) * 3 * file_count));
if (cal_files == NULL)
{
 printf("Failed to allocate memory for playback (%zu bytes)\n",
    sizeof(recording_t) * 3 * file_count);
 return 1;
}
memset(cal_files, 0, sizeof(recording_t) * 3 * file_count);
// Open each recording file and validate they were recorded in
   master/subordinate mode.
{
 // Reading the calibration files (box)
 std::vector<pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr>
    cal_cloud(file_count);
 for (size_t id = 0; id < file_count; id++)
 {
  //Open calibration video//
  cal_files[id].filename = "./"; // Only used to init the string
     var
  cal_files[id].filename = dir_Laser_input + std::string("SN1Cam"
     ) + std::to_string(id + 1) + std::string(".mkv");
  cal_files[id].playback = k4a::playback::open(cal_files[id].
     filename.c_str());
  if (!cal_files[id].playback.is_valid())
  {
   printf("Failed to open file: %s\n", cal_files[id].filename.
      c_str());
   break;
  }
  cal_files[id].record_config = cal_files[id].playback.
     get_record_configuration();
```

```
cal_files[id].k4a_calibration = cal_files[id].playback.
    get_calibration();
cal_files[id].camera_id = id;
cal_files[id].k4a_transform = k4a::transformation(cal_files[id
    ].k4a_calibration);
cal_files[id].playback.set_color_conversion(
    K4A_IMAGE_FORMAT_COLOR_BGRA32);
if (id == 0) {cal_files[id].record_config.wired_sync_mode =
    K4A_WIRED_SYNC_MODE_MASTER; printf("Opened subordinate
    recording file: %s\n", cal_files[id].filename.c_str());}
else if (cal_files[id].record_config.wired_sync_mode ==
    K4A_WIRED_SYNC_MODE_SUBORDINATE) printf("Opened subordinate
    recording file: %s\n", cal_files[id].filename.c_str());
else if (cal_files[id].record_config.wired_sync_mode ==
    K4A_WIRED_SYNC_MODE_STANDALONE) printf("Opened standalone
    recording file: %s\n", cal_files[id].filename.c_str());
else
{
 printf("ERROR: Failed to recognize the sync. mode: %s\n",
     cal_files[id].filename.c_str());
 result = K4A_RESULT_FAILED;
 break;
}
// Transform depth image to cloud
std::cout << "Capture for camera:" << id << std::endl;
cal_files[id].playback.seek_timestamp(std::chrono::duration<int
    , std::micro>(500000), K4A_PLAYBACK_SEEK_DEVICE_TIME);
if (!cal_files[id].playback.get_next_capture(&cal_files[id].
    capture))
 {
  printf("ERROR: Recording file is empty: %s\n", cal_files[id].
      filename.c_str());
 }
FrameInfo* frame = process_capture(&cal_files[id]);
if (frame->timestamp - std::chrono::duration<int, std::micro
    >(1000000).count() > 500000/ frame_rate) cal_files[id].
    playback.get_previous_capture(&cal_files[id].capture);
pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr source_cloud (new
```

```
    pcl::PointCloud<pcl::PointXYZRGBNormal>);
  pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr cloud (new pcl::
    PointCloud<pcl::PointXYZRGBNormal>);
  pcl::Indices indices;
  const int count = frame->ColorWidth * frame->ColorHeight;
  const int coord_stride = 1;
  const int rep_count = 100;
  source_cloud->reserve(count);
  std::vector<int> CloudBuf(frame->PointCloudData.size());
  std::vector<int> IRBuf(frame->IRImage.size());
  std::vector<int> valid(count, 0);
  std::cout <<"Generating Cloud From Frames, count ="<< count <<
      std::endl;
  // Averaging
  for(int rep_idx = 0; rep_idx < rep_count; rep_idx++){
   frame = process_capture(&cal_files[id]);
   int skip_from_top = 200;
   int skip_from_left = 100;
   int skip_from_right = 100;
   for (int i_px = skip_from_top * frame->ColorWidth; i_px <
      count; i_px++) {
    if(i_px % frame->ColorWidth < skip_from_left || i_px % frame
       ->ColorWidth > frame->ColorWidth - skip_from_right)
       continue;
    int XData_cast = (int) frame->PointCloudData.data()[i_px*3 ];
    int YData_cast = (int) frame->PointCloudData.data()[i_px*3 +
       1];
    int ZData_cast = (int) frame->PointCloudData.data()[i_px*3 +
       2];
    if(ZData_cast == 0) continue;
    CloudBuf[i_px*3] += XData_cast;
    CloudBuf[i_px*3 + 1] += YData_cast;
    CloudBuf[i_px*3 + 2] += ZData_cast;
    int IRData_cast = (int) frame->IRImage.data()[i_px];
    IRBuf[i_px] += IRData_cast;
    // valid[i_px] += (int) (ZData_cast != 0);
    valid[i_px] ++;
   }
```

```
    cal_files[id].playback.get_next_capture(&cal_files[id].capture
        );
  }
  // Pointcloud filling for Cal square
  for (int i_pt = 0; i_pt < count; i_pt += coord_stride) {
   if (valid[i_pt] == 0) continue;
   if (CloudBuf.data()[3 * i_pt + 2] / 1000.0f / (float)valid[
       i_pt] < 1.0f || CloudBuf.data()[3 * i_pt + 2] / 1000.0f / (
       float)valid[i_pt] > 3.0f) continue;
   if (IRBuf.data()[i_pt]/valid[i_pt] < 2000) continue;
   // if (id == 0|| IRBuf.data()[i_pt]/valid[i_pt] < 2500)
       continue;
   // BGR -> RGB
   pcl::PointXYZRGBNormal point;
   point.x = CloudBuf.data()[3 * i_pt + 0] / 1000.0f / (float)
       valid[i_pt];
   point.y = CloudBuf.data()[3 * i_pt + 1] / 1000.0f / (float)
       valid[i_pt];
   point.z = CloudBuf.data()[3 * i_pt + 2] / 1000.0f / (float)
       valid[i_pt];
   // point.r = (uint8_t)frame->ColorImage.data()[4 * i_pt + 2];
   // point.g = (uint8_t)frame->ColorImage.data()[4 * i_pt + 1];
   // point.b = (uint8_t)frame->ColorImage.data()[4 * i_pt + 0];
   point.r = (uint8_t)1;
   point.g = (uint8_t)1;
   point.b = (uint8_t)1;
   point.curvature = IRBuf.data()[i_pt]/ (float)valid[i_pt];
   if(std::isinf(point.x)||std::isinf(point.y)||std::isinf(point.
       z)) continue;
   if (point.r == 0 && point.g == 0 && point.b == 0) continue;
   source_cloud->points.push_back(point);
  }
  // Full cloud for validation
  for (int i_pt = 0; i_pt < count; i_pt += coord_stride) {
   if (valid[i_pt] == 0) continue;
   if (CloudBuf.data()[3 * i_pt + 2] / 1000.0f / (float)valid[
       i_pt] < 1.5f) continue;
   // BGR -> RGB
```

```
    pcl::PointXYZRGBNormal point;
    point.x = CloudBuf.data()[3 * i_pt + 0] / 1000.0f / (float)
        valid[i_pt];
    point.y = CloudBuf.data()[3 * i_pt + 1] / 1000.0f / (float)
        valid[i_pt];
    point.z = CloudBuf.data()[3 * i_pt + 2] / 1000.0f / (float)
        valid[i_pt];
    point.r = (uint8_t)frame->ColorImage.data()[4 * i_pt + 2];
    point.g = (uint8_t)frame->ColorImage.data()[4 * i_pt + 1];
    point.b = (uint8_t)frame->ColorImage.data()[4 * i_pt + 0];
    point.curvature = IRBuf.data()[i_pt]/ (float)valid[i_pt];
    if(std::isinf(point.x)||std::isinf(point.y)||std::isinf(point.
        z)) continue;
    if (point.r == 0 && point.g == 0 && point.b == 0) continue;
    cloud->points.push_back(point);
  }
  // Clustering the highly reflecting cloud to 3 groups and find
      the brightest point as the result.
  pcl::PointCloud<pcl::PointXYZRGBNormal> TempCloud;
  pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr StandardCloud(new
      pcl::PointCloud<pcl::PointXYZRGBNormal>);
  StandardCloud->points.push_back(pcl::PointXYZRGBNormal(0.0f,
      0.0f, 0.0f)); // A = origin
  StandardCloud->points.push_back(pcl::PointXYZRGBNormal(0.020f,
      0.0f, 0.0f)); // B = X-axis
  StandardCloud->points.push_back(pcl::PointXYZRGBNormal(0.0f,
      0.015f, 0.0f)); // C = Y-axis
  // Creating the KdTree object for the search method of the
      extraction
  pcl::search::KdTree<pcl::PointXYZRGBNormal>::Ptr tree (new pcl
      ::search::KdTree<pcl::PointXYZRGBNormal>);
  tree->setInputCloud (source_cloud);
  std::vector<pcl::PointIndices> cluster_indices;
  pcl::EuclideanClusterExtraction<pcl::PointXYZRGBNormal> ec;
  ec.setClusterTolerance (0.02); // 2cm
  ec.setMinClusterSize (5);
  ec.setMaxClusterSize (100);
  ec.setSearchMethod (tree);
```

```
ec.setInputCloud (source_cloud);
ec.extract (cluster_indices);
int j = 0;
for (const auto& cluster : cluster_indices)
{
pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr cloud_cluster (new
    pcl::PointCloud<pcl::PointXYZRGBNormal>);
for (const auto& idx : cluster.indices) {
cloud_cluster->push_back((*source_cloud)[idx]);
} //*
cloud_cluster->width = cloud_cluster->size ();
cloud_cluster->height = 1;
cloud_cluster->is_dense = true;
std::cout << "PointCloud representing the Cluster: " <<
   cloud_cluster->size () << " data points." << std::endl;
std::vector<float> brightness(cloud_cluster->size(), 0);
for(int i_px = 0; i_px < cloud_cluster->size(); i_px++)
   brightness[i_px] = cloud_cluster->points[i_px].curvature;
std::vector<float>::iterator id_imax = std::max_element(
   brightness.begin(), brightness.end());
size_t id_max = std::distance(brightness.begin(), id_imax);
pcl::PointXYZRGBNormal MKpoint = cloud_cluster->points[id_max];
float MKdistance = std::sqrt(std::pow(MKpoint.x, 2) + std::pow(
   MKpoint.y, 2) + std::pow(MKpoint.z, 2));
MKpoint.x += 0.007* MKpoint.x / MKdistance;
MKpoint.y += 0.007* MKpoint.y / MKdistance;
MKpoint.z += 0.007* MKpoint.z / MKdistance;
MKpoint.r = 0;
MKpoint.g = 0;
MKpoint.b = 0;
TempCloud.push_back(MKpoint);
j++;
}
if(j < 3) std::cout << "Warning!! The bright clusters are less
   than 3!!\n Enlarge the sizes of the frames.";
if(j > 3) std::cout << "Warning!! The bright clusters are more
   than 4!!\n Reduce the sizes of the frames.";
std::vector<size_t> order_vertice(3, 0);
```

```
    std::vector<float> distances(3, 0);
    for(int id_d = 0; id_d < 3; id_d++){
     if(distances[id_d] > 0.23f){order_vertice[id_d] = 0; continue;
     }else if(distances[id_d] < 0.18f){order_vertice[id_d] = 1;
        continue;
     }else{order_vertice[id_d] = 2;}
    }
    std::cout << order_vertice[0] << "-" << distances[0] << std::
       endl;
    std::cout << order_vertice[1] << "-" << distances[1] << std::
       endl;
    std::cout << order_vertice[2] << "-" << distances[2] << std::
       endl;
    pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr SquareCloud(new
       pcl::PointCloud<pcl::PointXYZRGBNormal>);
    for(int id_d = 0; id_d < 3; id_d++) if(order_vertice[id_d] ==
       0) SquareCloud->push_back(TempCloud.points[id_d]);
    for(int id_d = 0; id_d < 3; id_d++) if(order_vertice[id_d] ==
       1) SquareCloud->push_back(TempCloud.points[id_d]);
    for(int id_d = 0; id_d < 3; id_d++) if(order_vertice[id_d] ==
       2) SquareCloud->push_back(TempCloud.points[id_d]);
#ifdef ManualRefine
    if(id==0) {(*SquareCloud)[0].y -= 0.01;(*SquareCloud)[2].y +=
       0.005;}
    if(id==1) {(*SquareCloud)[1].x += 0.015;(*SquareCloud)[1].y +=
       0.01;}
    if(id==2) {(*SquareCloud)[0].y -= 0.005;(*SquareCloud)[1].y +=
       0.01;}
#endif
    SquareCloud->points[0].r = 255;
    SquareCloud->points[1].g = 255;
    SquareCloud->points[2].b = 255;
    files[id].laserCalCloud = SquareCloud;
    cal_cloud[id] = cloud;
    frame->CameraIndex = id;
    frame->filename = cal_files[id].filename;
    // Resource recycling;
    cal_files[id].capture.reset();
```

```
    Eigen::Matrix4f trans2mas_e = Eigen::Matrix4f::Identity();
    pcl::registration::TransformationEstimationDualQuaternion<pcl::
        PointXYZRGBNormal, pcl::PointXYZRGBNormal> trans;
    trans.estimateRigidTransformation(*SquareCloud, *StandardCloud,
         trans2mas_e);
    pcl::transformPointCloudWithNormals(*files[id].laserCalCloud, *
        files[id].laserCalCloud, trans2mas_e);
    pcl::transformPointCloudWithNormals(*source_cloud, *
        source_cloud, trans2mas_e);
    pcl::transformPointCloudWithNormals(*cloud, *cloud,
        trans2mas_e);
    files[id].transform2master << Eigen::Matrix4f(trans2mas_e);
    std::string filename = "../output/filteredcloud_" + std::
        to_string(id) + ".pcd";
    pcl::io::savePCDFileBinary(filename, *source_cloud);
    filename = "../output/SqrCalCloud_" + std::to_string(id) + ".
        pcd";
    pcl::io::savePCDFileBinary(filename, *files[id].laserCalCloud);
    filename = "../output/CalCloud_" + std::to_string(id) + ".pcd";
    pcl::io::savePCDFileBinary(filename, *cloud);
  }
  for(int id = 0; id < file_count; id++){
    std::string matname2 = "Laser_transform" + std::to_string(id) +
         ".txt";
    std::ofstream fout;
    fout.open(matname2, std::ios::out); // Overwrite the transform
        file
    fout << files[id].transform2master << std::endl;
    std::cout <<"mat txt saved"<< std::endl;
    fout.close();
  }
}
std::cout <<"Calibration matrix extracted.\nMove to the process
    fitting the Z axis to upward and X axis to LoS roughly."<< std
    ::endl;
std::cout <<"========================"<< std::endl;
// Main process to transform the images to the point clouds./
if (result != K4A_RESULT_SUCCEEDED) {
```

```
 printf("Some error occured in initiation process!");
 return -1;
}
printf("%-32s - %12s - %12s - %12s\n", "Source file", "COLOR", "
   DEPTH", "IR");
printf("=========================\n");
bool terminated = false;
double count = 0;
//Open test video//
for (size_t i = 0; i < file_count; i++)
{
 files[i].filename = "./"; // Only used to init the string var
 std::string dir_test_input;
 dir_test_input = std::string("../../../THzExp/") + std::string(
    argv[1]) + std::string("/SquareCal/");
 files[i].filename = dir_test_input + std::string("SN1Cam") + std
    ::to_string(i + 1) + std::string(".mkv");
 files[i].playback = k4a::playback::open(files[i].filename.c_str
    ());
 std::cout << files[i].filename << std::endl;
 if (!files[i].playback.is_valid())
 {
  printf("Failed to open file: %s\n", files[i].filename.c_str());
  break;
 }
 files[i].record_config = files[i].playback.
    get_record_configuration();
 files[i].k4a_calibration = files[i].playback.get_calibration();
 files[i].camera_id = i;
 files[i].k4a_transform = k4a::transformation(files[i].
    k4a_calibration);
 files[i].playback.set_color_conversion(
    K4A_IMAGE_FORMAT_COLOR_BGRA32);
 if (i == 0) {files[i].record_config.wired_sync_mode =
    K4A_WIRED_SYNC_MODE_MASTER; printf("Opened master recording
    file: %s\n", files[i].filename.c_str());}
 else if (files[i].record_config.wired_sync_mode ==
    K4A_WIRED_SYNC_MODE_SUBORDINATE) printf("Opened subordinate
```

```
    recording file:%s\n", files[i].filename.c_str());
  else if (files[i].record_config.wired_sync_mode ==
    K4A_WIRED_SYNC_MODE_STANDALONE) printf("Opened standalone
    recording file:%s\n", files[i].filename.c_str());
  else
  {
   printf("ERROR: Failed to recognize the sync. mode:%s\n", files
     [i].filename.c_str());
   result = K4A_RESULT_FAILED;
   break;
  }
  // Find the nearest frame to the assigned time stamp.
  files[i].playback.seek_timestamp(std::chrono::duration<int, std
    ::micro>(1000000), K4A_PLAYBACK_SEEK_DEVICE_TIME);
  if (!files[i].playback.get_next_capture(&files[i].capture))
  {
  printf("ERROR: Recording file is empty:%s\n", files[i].filename
    .c_str());
  }
  FrameInfo* frame = process_capture(&files[i]);
  std::cout << frame->timestamp << std::endl;
  if (frame->timestamp - std::chrono::duration<int, std::micro
    >(1000000).count() > 500000/ frame_rate) files[i].playback.
    get_previous_capture(&files[i].capture);
}
while (!terminated) {
 count++;
 ExtrinsicsCalibration extrinsicsCalib;
 std::vector<FrameInfo*> frames(file_count);
 std::vector<AlignmentTransform> extrinsics(file_count);
 std::vector<uint64_t> timestamps(file_count);
 k4a_stream_result_t stream_result;
 // Find the lowest timestamp out of each of the current captures
    .
 for (size_t i = 0; i < file_count; i++)
 {
  if (files[i].playback.get_next_capture(&files[i].capture))
  {
```

```
    std::cout << "Capture for camera:" << i << std::endl;
    FrameInfo* frame = process_capture(&files[i]);
    printf("Capture time: %ld\n", frame->timestamp);
    k4a_imu_sample_t imu_sample;
    files[i].playback.get_next_imu_sample(&imu_sample);
    for (int j = 0; j < 3; ++j) {
     frame->Accelerometer[j] = imu_sample.acc_sample.v[j];
    }
    frame->CameraIndex = i;
    frame->filename = files[i].filename;
    frames[i] = frame;
    timestamps[i]= frame->timestamp;
    extrinsics[i] = frame->transform2master;
    files[i].capture.reset();
   }
   else {
    terminated = true;
    break;
   }
  }
  std::cout << "Calculating Extrisincs" << std::endl;
  if (!extrinsicsCalib.CalculateExtrinsics(frames, extrinsics,
     timestamps)) {
   std::cout << "Full registration failed" << std::endl;
  }
  for(auto& frm : frames) delete frm;
  break;
 }
 for (size_t i = 0; i < file_count; i++)
 {
  if (files[i].handle != NULL)
  {
   k4a_playback_close(files[i].handle);
   files[i].handle = NULL;
  }
 }
 free(files);
 return result == K4A_RESULT_SUCCEEDED ? 0 : 1;
```

```
}
```

Listing B.5: include/camera_extrinsics.cpp

```
#include "camera_extrinsics.hpp"
#include <fstream>
#include <filesystem>
#include <typeinfo>
namespace fs = std::filesystem;
bool ExtrinsicsCalibration::GenerateFullCloudFromFrames(
 FrameInfo frame)
{
 const int idx = frame.CameraIndex;
 if (!frame.PointCloudData.data()) {
  return false;
 }
 pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr source_cloud (new
    pcl::PointCloud<pcl::PointXYZRGBNormal>);
 pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr cloud (new pcl::
    PointCloud<pcl::PointXYZRGBNormal>);
 pcl::Indices indices;
 full_cloud[idx] = cloud;
 const int count = frame.ColorWidth * frame.ColorHeight;
 const int coord_stride = 1;
 source_cloud->reserve(count);
 // full_cloud[idx]
 std::cout <<"Generating Cloud From Frames, count ="<< count <<
    std::endl;
 for (int i = 0; i < count; i += coord_stride) {
  if (frame.PointCloudData.data()[3 * i + 2] == 0) continue;
  // BGR -> RGB
  pcl::PointXYZRGBNormal point;
  point.x = frame.PointCloudData.data()[3 * i + 0] / 1000.0f;
  point.y = frame.PointCloudData.data()[3 * i + 1] / 1000.0f;
  point.z = frame.PointCloudData.data()[3 * i + 2] / 1000.0f;
  point.r = (uint8_t)frame.ColorImage.data()[4 * i + 2];
  point.g = (uint8_t)frame.ColorImage.data()[4 * i + 1];
  point.b = (uint8_t)frame.ColorImage.data()[4 * i + 0];
  if (point.r == 0 && point.g == 0 && point.b == 0) continue;
```

```
  source_cloud->points.push_back(point);
 }
 pcl::StatisticalOutlierRemoval<pcl::PointXYZRGBNormal> sor;
 sor.setInputCloud (source_cloud);
 sor.setMeanK (40);
 sor.setStddevMulThresh (1.0);
 sor.filter (*cloud);
 pcl::removeNaNFromPointCloud(*cloud, *cloud, indices);
 std::cout <<"Generated Cloud From Frames, count ="<< full_cloud[
    idx]->size() << std::endl;
 pcl::NormalEstimationOMP<pcl::PointXYZRGBNormal, pcl::
    PointXYZRGBNormal> ne;
 const double normal_radius = 0.05;
 ne.setInputCloud (cloud);
 // Create an empty kdtree representation, and pass it to the
    normal estimation object.
 // Its content will be filled inside the object, based on the
    given input dataset (as no other search surface is given).
 pcl::search::Octree<pcl::PointXYZRGBNormal>::Ptr tree (new pcl::
    search::Octree<pcl::PointXYZRGBNormal>(normal_radius));
 ne.setSearchMethod (tree);
 // Use all neighbors in a sphere of radius 3cm
 ne.setRadiusSearch (normal_radius);
 ne.useSensorOriginAsViewPoint();
 ne.compute (*cloud);
 std::cout <<"Constructed PCL PointCloud for Camera ID: " << idx
    <<std::endl;
 return true;
}
bool ExtrinsicsCalibration::GenerateBodyCloudFromFrames(
 FrameInfo frame)
{
 const int idx = frame.CameraIndex;
 if (!frame.PointCloudData.data()) {
  return false;
 }
 pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr source_cloud (new
    pcl::PointCloud<pcl::PointXYZRGBNormal>);
```

```
pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr cloud (new pcl::
    PointCloud<pcl::PointXYZRGBNormal>);
pcl::Indices indices;
full_cloud[idx] = cloud;
const int count = frame.ColorWidth * frame.ColorHeight;
const int coord_stride = 1;
source_cloud->reserve(count);
std::cout << frame.idx_closest_body << std::endl;
for (int i = 0; i < count; i += coord_stride) {
 if (frame.PointCloudData.data()[3 * i + 2] == 0 || frame.
     BodyMapImage.data()[i] != frame.idx_closest_body|| frame.
     PointCloudData.data()[3 * i + 1] > 900.0f ||frame.
     PointCloudData.data()[3 * i + 1] < -1700.0f) continue;
 // BGR -> RGB
 pcl::PointXYZRGBNormal point;
 point.x = frame.PointCloudData.data()[3 * i + 0] / 1000.0f;
 point.y = frame.PointCloudData.data()[3 * i + 1] / 1000.0f;
 point.z = frame.PointCloudData.data()[3 * i + 2] / 1000.0f;
 point.r = (uint8_t)frame.ColorImage.data()[4 * i + 2];
 point.g = (uint8_t)frame.ColorImage.data()[4 * i + 1];
 point.b = (uint8_t)frame.ColorImage.data()[4 * i + 0];
 if (point.r == 0 && point.g == 0 && point.b == 0) continue;
 source_cloud->points.push_back(point);
}
pcl::StatisticalOutlierRemoval<pcl::PointXYZRGBNormal> sor;
sor.setInputCloud (source_cloud);
sor.setMeanK (40);
sor.setStddevMulThresh (1.0);
sor.filter (*cloud);
pcl::removeNaNFromPointCloud(*cloud, *cloud, indices);
std::cout <<"Generated Cloud From Frames, count ="<< full_cloud[
    idx]->size() << std::endl;
pcl::NormalEstimationOMP<pcl::PointXYZRGBNormal, pcl::
    PointXYZRGBNormal> ne;
const double normal_radius = 0.01;
ne.setInputCloud (cloud);
// Create an empty kdtree representation, and pass it to the
    normal estimation object.
```

```
 // Its content will be filled inside the object, based on the
    given input dataset (as no other search surface is given).
 pcl::search::Octree<pcl::PointXYZRGBNormal>::Ptr tree (new pcl::
    search::Octree<pcl::PointXYZRGBNormal>(normal_radius));
 ne.setSearchMethod (tree);
 // Use all neighbors in a sphere of radius 3cm
 ne.setRadiusSearch (normal_radius);
 ne.useSensorOriginAsViewPoint();
 ne.compute (*cloud);
 std::cout <<"Constructed PCL PointCloud for Camera ID: " << idx
    <<std::endl;
 // std::string filename = "../output/fulcloud_" + std::to_string(
    idx) + ".pcd";
 return true;
}
bool ExtrinsicsCalibration::GenerateBGCloudFromFrames(
 FrameInfo frame)
{
 const int idx = frame.CameraIndex;
 if (!frame.PointCloudData.data()) {
  return false;
 }
 pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr source_cloud (new
    pcl::PointCloud<pcl::PointXYZRGBNormal>);
 pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr cloud (new pcl::
    PointCloud<pcl::PointXYZRGBNormal>);
 pcl::Indices indices;
 full_cloud[idx] = cloud;
 const int count = frame.ColorWidth * frame.ColorHeight;
 const int coord_stride = 1;
 source_cloud->reserve(count);
 std::cout <<"Generating Cloud From Frames, count = "<< count <<
    std::endl;
 for (int i = 0; i < count; i += coord_stride) {
  if (frame.PointCloudData.data()[3 * i + 2] == 0 || frame.
     BodyMapImage.data()[i] != K4ABT_BODY_INDEX_MAP_BACKGROUND ||
     frame.PointCloudData.data()[3 * i + 1] > 900.0f ||frame.
     PointCloudData.data()[3 * i + 1] < -1700.0f) continue;
```

```
  // BGR -> RGB
  pcl::PointXYZRGBNormal point;
  point.x = frame.PointCloudData.data()[3 * i + 0] / 1000.0f;
  point.y = frame.PointCloudData.data()[3 * i + 1] / 1000.0f;
  point.z = frame.PointCloudData.data()[3 * i + 2] / 1000.0f;
  point.r = (uint8_t)frame.ColorImage.data()[4 * i + 2];
  point.g = (uint8_t)frame.ColorImage.data()[4 * i + 1];
  point.b = (uint8_t)frame.ColorImage.data()[4 * i + 0];
  if (point.r == 0 && point.g == 0 && point.b == 0) continue;
  source_cloud->points.push_back(point);
 }
 pcl::StatisticalOutlierRemoval<pcl::PointXYZRGBNormal> sor;
 sor.setInputCloud (source_cloud);
 sor.setMeanK (40);
 sor.setStddevMulThresh (1.0);
 sor.filter (*cloud);
 pcl::removeNaNFromPointCloud(*cloud, *cloud, indices);
 std::cout <<"Generated Cloud From Frames, count ="<< full_cloud[
    idx]->size() << std::endl;
 std::cout <<"Constructed PCL PointCloud for Camera ID: " << idx
    <<std::endl;
 return true;
}
bool ExtrinsicsCalibration::DownSample(pcl::PointCloud<pcl::
   PointXYZRGBNormal>::Ptr& cloud, const float voxel_size, const
   int idx)
{
 pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr cloud_filtered(new
    pcl::PointCloud<pcl::PointXYZRGBNormal>);
 pcl::Indices indices;
 // Downsample the point cloud
 pcl::VoxelGrid<pcl::PointXYZRGBNormal> sor;
 sor.setInputCloud (full_cloud[idx]);
 sor.setLeafSize (voxel_size, voxel_size,voxel_size);
 sor.filter(*cloud_filtered);
 cloud = cloud_filtered;
 if (!cloud) {
  std::cout <<"VoxelDownSample failed"<< std::endl;
```

```
  return false;
 }
 // Estimate normals with full resolution point cloud
 pcl::NormalEstimationOMP<pcl::PointXYZRGBNormal, pcl::
    PointXYZRGBNormal> ne;
 const double normal_radius = voxel_size * 5.0;
 ne.setInputCloud (cloud_filtered);
 // Create an empty kdtree representation, and pass it to the
    normal estimation object.
 // Its content will be filled inside the object, based on the
    given input dataset (as no other search surface is given).
 pcl::search::Octree<pcl::PointXYZRGBNormal>::Ptr tree (new pcl::
    search::Octree<pcl::PointXYZRGBNormal>(voxel_size));
 ne.setSearchMethod (tree);
 // Use all neighbors in a sphere of radius 3cm
 ne.setRadiusSearch (normal_radius);
 ne.useSensorOriginAsViewPoint();
 ne.compute (*cloud);
 pcl::removeNaNFromPointCloud(*cloud, *cloud, indices);
 indices.clear();
 pcl::removeNaNNormalsFromPointCloud(*cloud, *cloud, indices);
 return true;
}
bool ExtrinsicsCalibration::CalculateExtrinsics(
 const std::vector<FrameInfo*>& frames,
 std::vector<AlignmentTransform>& output,
 std::vector<uint64_t>& timestamps)
{
 if (frames.empty()) {
  std::cout  <<"No images provided to registration"<< std::endl;
  return false;
 }
 const int camera_count = static_cast<int>( frames.size() );
 output.resize(camera_count);
 full_cloud.resize(camera_count);
 std::vector<Eigen::Matrix4f> current_transform(camera_count);
 std::vector<Eigen::Matrix4f> final_transform(camera_count);
 for (int camera_index = 0; camera_index < camera_count; ++
```

```
    camera_index)
{
 GenerateFullCloudFromFrames(*(frames[camera_index]));
 std::ifstream fin;
 std::string matfile = "Laser_transform" + std::to_string(
    camera_index)+ ".txt";
 fin.open(matfile);
 Eigen::Matrix4d temp;
 for(int i =0;i<4;i++){
  for(int j =0;j<4;j++){
    fin >> temp(i,j);
  }
 }
 current_transform[camera_index]=temp.cast<float>();
 frames[camera_index]->transform2master = current_transform[
    camera_index];
 fin.close();
}
for(int camera_index = 0; camera_index < camera_count; ++
   camera_index) {
 output[camera_index] = current_transform[camera_index].cast<
    float>();
 final_transform[camera_index] = current_transform[camera_index].
    cast<float>();
}
pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr final_cloud(new pcl
   ::PointCloud<pcl::PointXYZRGBNormal>);
std::cout <<"Saving point cloud"<< std::endl;
for (int camera_index = 0; camera_index < camera_count; ++
   camera_index)
{
 pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr cloud_i(full_cloud[
    camera_index]);
 pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr cloud_out(new pcl::
    PointCloud<pcl::PointXYZRGBNormal>);
 pcl::transformPointCloudWithNormals(*cloud_i, *cloud_out,
    frames[camera_index]->transform2master);
 std::string filename = "../output/cloud_" + std::to_string(
```

```
      camera_index) + ".pcd";
  *final_cloud += *cloud_out;
  cloud_out->clear();
 }
 #ifdef BOXCLIP
  pcl::PassThrough<pcl::PointXYZRGBNormal> pass;
  pass.setInputCloud (final_cloud);
  pass.setFilterFieldName ("x");
  pass.setFilterLimits (-2.2f, 2.2f);
  pass.filter (*final_cloud);
  pass.setFilterFieldName ("y");
  pass.setFilterLimits (-2.2f, 2.2f);
  pass.filter (*final_cloud);
  pass.setFilterFieldName ("z");
  pass.setFilterLimits (-3.0f, 1.5f);
  pass.filter (*final_cloud);
 #endif
 std::filesystem::create_directory("../output/finalbodycloud");
 std::string filename = "../output/finalbodycloud/" +std::
     to_string(timestamps[0])+ ".pcd";
 pcl::io::savePCDFileBinary(filename, *final_cloud);
 std::cout <<"Point cloud saved, time (usec) was:"<< std::
     to_string(timestamps[0]) << std::endl;
 std::cout <<"
     ==========================================================="<<
      std::endl;
 for(auto& cld : full_cloud) {
  cld->clear();
 }
 full_cloud.clear();
 final_cloud->clear();
 return true;
}
```

B.3.2 Codes for generating PC-represented human bodies

Be advised that the contents are different from the files desplayed in last section though the file names are the same. The directory tree is as follows,

```
—SquareReg
  |--build
  | |--calib_k4a (Main program)
  |--CalMatriceArc (Input transformation matrices)
  | |--yyyymmdd (Token for identifing matrices for different
     measurement.)
  | | |--Laser_transform0.txt
  | | |--Laser_transform1.txt
  | | |--Laser_transform2.txt
  | | |--Laser_transform3.txt
  |--include (Header files)
  | |--camera_extrinsics.hpp
  | |--LaserCalib.hpp
  |--input
  | |--yyyymmdd (Measurement date for identification)
  |--output (Storage for output point clouds)
  |--src (Source codes)
  | |--camera_extrinsics.cpp
  | |--LaserCalib.cpp
  |--CMakelists.txt
```

Listing B.6: CMakelists.txt

```
cmake_minimum_required(VERSION 3.15)
project(calib_k4a LANGUAGES CXX)
set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
find_package(k4a REQUIRED)
find_package(k4abt REQUIRED)
find_package(k4arecord REQUIRED)
find_package(PCL 1.12 REQUIRED)
# Source
set(INCLUDE_FILES
  include/camera_extrinsics.hpp
  include/LaserCalib.hpp
)
set(SOURCE_FILES
  ${INCLUDE_FILES}
```

```
  src/camera_extrinsics.cpp
  src/LaserCalib.cpp
)
set(EXENAME
  laser_reg
)
include_directories(${PCL_INCLUDE_DIRS})
link_directories(${PCL_LIBRARY_DIRS})
add_definitions(${PCL_DEFINITIONS})
# Targets
add_executable(${EXENAME} ${SOURCE_FILES})
target_include_directories(${EXENAME} PUBLIC include)
target_link_libraries(${EXENAME} PUBLIC
  k4a # Kinect SDK
  k4abt
  k4arecord
  ${PCL_LIBRARIES}
  stdc++fs # filesystem
)
```

Listing B.7: include/LaserCalib.hpp

```
#pragma once
#include <fstream>
#include <filesystem>
#include "camera_extrinsics.hpp"
#include <k4arecord/playback.hpp>
#include <pcl/registration/
    transformation_estimation_dual_quaternion.h>
// Customized structure
typedef struct
{
  std::string filename;
  std::string out_dir;
  k4a_playback_t handle;
  k4a::playback playback;
  k4a_record_configuration_t record_config;
  k4a::calibration k4a_calibration;
  k4a::transformation k4a_transform;
```

```
  Eigen::Quaternionf rotation2master;
  Eigen::Matrix4f transform2master;
  k4a::capture capture;
  k4abt_body_t body;
  int idx_closest_body;
  std::vector<bool> conf_lev;
  pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr laserCalCloud;
  std::vector<Eigen::Vector3f> frame_cloud;
  int camera_id;
} recording_t;
// Declearation of the functions
void print_calibration(k4a::calibration& calibration);
uint64_t first_capture_timestamp(k4a_capture_t capture);
uint64_t capture_depth_timestamp(k4a::capture capture);
FrameInfo* process_capture_incbody(recording_t *file);
FrameInfo* process_capture(recording_t *file);
pcl::PointXYZRGBNormal Laser_coord_exract(recording_t *file, int
   reptime, std::chrono::duration<int, std::micro> laser_time, int
    color);
void CopyIntrinsics(const k4a_calibration_camera_t& from,
   CameraIntrinsics& to);
void print_body_information(k4abt_body_t body);
// Definition of the functions
void print_calibration(k4a::calibration& calibration)
  {
    {
      std::cout << "Depth camera:" << std::endl;
      auto calib = calibration.depth_camera_calibration;
      std::cout << "resolution width:" << calib.resolution_width
        << std::endl;
      std::cout << "resolution height:" << calib.
         resolution_height << std::endl;
      std::cout << "principal point x:" << calib.intrinsics.
         parameters.param.cx << std::endl;
      std::cout << "principal point y:" << calib.intrinsics.
         parameters.param.cy << std::endl;
      std::cout << "focal length x:" << calib.intrinsics.
         parameters.param.fx << std::endl;
```

```
    std::cout << "focal length y: " << calib.intrinsics.
        parameters.param.fy << std::endl;
    std::cout << "radial distortion coefficients:" << std::endl;
    std::cout << "k1: " << calib.intrinsics.parameters.param.k1
        << std::endl;
    std::cout << "k2: " << calib.intrinsics.parameters.param.k2
        << std::endl;
    std::cout << "k3: " << calib.intrinsics.parameters.param.k3
        << std::endl;
    std::cout << "k4: " << calib.intrinsics.parameters.param.k4
        << std::endl;
    std::cout << "k5: " << calib.intrinsics.parameters.param.k5
        << std::endl;
    std::cout << "k6: " << calib.intrinsics.parameters.param.k6
        << std::endl;
    std::cout << "center of distortion in Z=1 plane, x: " <<
        calib.intrinsics.parameters.param.codx << std::endl;
    std::cout << "center of distortion in Z=1 plane, y: " <<
        calib.intrinsics.parameters.param.cody << std::endl;
    std::cout << "tangential distortion coefficient x: " <<
        calib.intrinsics.parameters.param.p1 << std::endl;
    std::cout << "tangential distortion coefficient y: " <<
        calib.intrinsics.parameters.param.p2 << std::endl;
    std::cout << "metric radius: " << calib.intrinsics.
        parameters.param.metric_radius << std::endl;
}
{
    std::cout << "Color camera:" << std::endl;
    auto calib = calibration.color_camera_calibration;
    std::cout << "resolution width: " << calib.resolution_width
        << std::endl;
    std::cout << "resolution height: " << calib.
        resolution_height << std::endl;
    std::cout << "principal point x: " << calib.intrinsics.
        parameters.param.cx << std::endl;
    std::cout << "principal point y: " << calib.intrinsics.
        parameters.param.cy << std::endl;
    std::cout << "focal length x: " << calib.intrinsics.
```

```
        parameters.param.fx << std::endl;
    std::cout << "focal length y: " << calib.intrinsics.
        parameters.param.fy << std::endl;
    std::cout << "radial distortion coefficients:" << std::endl;
    std::cout << "k1: " << calib.intrinsics.parameters.param.k1
        << std::endl;
    std::cout << "k2: " << calib.intrinsics.parameters.param.k2
        << std::endl;
    std::cout << "k3: " << calib.intrinsics.parameters.param.k3
        << std::endl;
    std::cout << "k4: " << calib.intrinsics.parameters.param.k4
        << std::endl;
    std::cout << "k5: " << calib.intrinsics.parameters.param.k5
        << std::endl;
    std::cout << "k6: " << calib.intrinsics.parameters.param.k6
        << std::endl;
    std::cout << "center of distortion in Z=1 plane, x: " <<
        calib.intrinsics.parameters.param.codx << std::endl;
    std::cout << "center of distortion in Z=1 plane, y: " <<
        calib.intrinsics.parameters.param.cody << std::endl;
    std::cout << "tangential distortion coefficient x: " <<
        calib.intrinsics.parameters.param.p1 << std::endl;
    std::cout << "tangential distortion coefficient y: " <<
        calib.intrinsics.parameters.param.p2 << std::endl;
    std::cout << "metric radius: " << calib.intrinsics.
        parameters.param.metric_radius << std::endl;
}
auto extrinsics = calibration.extrinsics[
    K4A_CALIBRATION_TYPE_DEPTH][K4A_CALIBRATION_TYPE_COLOR];
std::cout << "depth2color translation: (" << extrinsics.
    translation[0] << "," << extrinsics.translation[1] << ","
    << extrinsics.translation[2] << ")" << std::endl;
std::cout << "depth2color rotation: |" << extrinsics.rotation
    [0] << "," << extrinsics.rotation[1] << "," << extrinsics.
    rotation[2] << "|" << std::endl;
std::cout << "------------|" << extrinsics.rotation[3] << ","
    << extrinsics.rotation[4] << "," << extrinsics.rotation[5]
    << "|" << std::endl;
```

```
    std::cout << "------------|" << extrinsics.rotation[6] << ","
      << extrinsics.rotation[7] << "," << extrinsics.rotation[8]
      << "|" << std::endl;
  }
uint64_t first_capture_timestamp(k4a_capture_t capture)
{
  uint64_t min_timestamp = (uint64_t)-1;
  k4a_image_t images[3];
  images[0] = k4a_capture_get_color_image(capture);
  images[1] = k4a_capture_get_depth_image(capture);
  images[2] = k4a_capture_get_ir_image(capture);
  for (int i = 0; i < 3; i++)
  {
    if (images[i] != NULL)
    {
      uint64_t timestamp = k4a_image_get_device_timestamp_usec(
        images[i]);
      if (timestamp < min_timestamp)
      {
        min_timestamp = timestamp;
      }
      k4a_image_release(images[i]);
      images[i] = NULL;
    }
  }
  return min_timestamp;
}
uint64_t capture_depth_timestamp(k4a::capture capture)
{
  k4a::image image;
  uint64_t timestamp = 0;
  image = capture.get_depth_image();
  if (image != NULL)
  {
    uint64_t timestamp = image.get_device_timestamp().count();
    image.reset();
  }
  return timestamp;
```

```
}
FrameInfo* process_capture_incbody(recording_t *file)
{
  k4a::image images[3];
  FrameInfo* frame = new FrameInfo();
  images[0] = file->capture.get_color_image();
  images[1] = file->capture.get_depth_image();
  k4abt::tracker tracker = k4abt::tracker::create(file->
     k4a_calibration);
  if (!tracker.enqueue_capture(file->capture))
  {
    // It should never hit timeout when K4A_WAIT_INFINITE is set.
    std::cout << "Error! Add capture to tracker process queue
       timeout!" << std::endl;
  }
  k4abt::frame body_frame = tracker.pop_result();
  tracker.shutdown();
  if (body_frame != nullptr)
  {
    uint32_t num_bodies = body_frame.get_num_bodies();

    if(num_bodies)
    {
      int idx_closest_body = 0;
      for(int idx_body = 0; idx_body < num_bodies; idx_body++) {
        k4abt_body_t closest_body = body_frame.get_body(
           idx_closest_body);
        k4abt_body_t current_body = body_frame.get_body(idx_body);
        if(closest_body.skeleton.joints[0].position.v[2] >
           current_body.skeleton.joints[0].position.v[2])
           idx_closest_body = idx_body;
      }
      k4abt_body_t body = body_frame.get_body(idx_closest_body);
      k4abt_joint_t joint = body.skeleton.joints[3];
      frame->neck.push_back(pcl::PointXYZRGBNormal((body.skeleton.
         joints[3].position.v[0]/1000.0f)
          , (body.skeleton.joints[3].position.v[1]/1000.0f)
          , (body.skeleton.joints[3].position.v[2]/1000.0f)));
```

```
      frame->confLevel = body.skeleton.joints[K4ABT_JOINT_NECK].
          confidence_level;
      images[2] = body_frame.get_body_index_map();
      frame->idx_closest_body = idx_closest_body;
      #ifdef CHECKJOINTS
        pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr cloud_joints(
            new pcl::PointCloud<pcl::PointXYZRGBNormal>);
        for(int idx_joint = 0; idx_joint < 32; idx_joint++) {
          cloud_joints->push_back(pcl::PointXYZRGBNormal((body.
              skeleton.joints[idx_joint].position.v[0]/1000.0f)
          , (body.skeleton.joints[idx_joint].position.v[1]/1000.0f
              )
          , (body.skeleton.joints[idx_joint].position.v[2]/1000.0f
              )));
        }
          std::string filename = file->out_dir + "/joints/" +std
              ::to_string(images[1].get_device_timestamp().count()
              - (int64_t) file->record_config.
              start_timestamp_offset_usec)+"Cam"+std::to_string(
              file->camera_id) +".pcd";
          pcl::io::savePCDFileBinary(filename, *cloud_joints);
      #endif
    }
  }
  // Copy depth
  frame->DepthWidth = images[1].get_width_pixels();
  frame->DepthHeight = images[1].get_height_pixels();
  frame->DepthStride = images[1].get_stride_bytes();
  const unsigned depth_size = frame->DepthStride * frame->
      DepthHeight;
  frame->DepthImage.resize(depth_size);
  memcpy(frame->DepthImage.data(), reinterpret_cast<const uint16_t
      *>( images[1].get_buffer()), depth_size * 2);
  // Copy body map
  if(images[2].is_valid()){
    frame->BodyMapWidth = images[2].get_width_pixels();
    frame->BodyMapHeight = images[2].get_height_pixels();
    frame->BodyMapStride = images[2].get_stride_bytes();
```

```
    const unsigned bodymap_size = frame->BodyMapStride * frame->
        BodyMapHeight;
    frame->BodyMapImage.resize(bodymap_size);
    memcpy(frame->BodyMapImage.data(), reinterpret_cast<const
        uint8_t*>( images[2].get_buffer()), bodymap_size);
}
k4a::image transformed_color_image = file->k4a_transform.
    color_image_to_depth_camera(images[1], images[0]);
k4a::image point_cloud_image = file->k4a_transform.
    depth_image_to_point_cloud(images[1],
    K4A_CALIBRATION_TYPE_DEPTH);
// Copy point cloud
const size_t cloud_buf_size = point_cloud_image.get_size();
frame->PointCloudData.resize(cloud_buf_size * 2);
memcpy(frame->PointCloudData.data(), reinterpret_cast<const
    int16_t*>( point_cloud_image.get_buffer() ), cloud_buf_size )
    ;
// Copy color
frame->ColorWidth = transformed_color_image.get_width_pixels();
frame->ColorHeight = transformed_color_image.get_height_pixels()
    ;
frame->ColorStride = transformed_color_image.get_stride_bytes();
const uint8_t* color_image = \
  reinterpret_cast<const uint8_t*>( transformed_color_image.
      get_buffer());
const size_t color_size = transformed_color_image.get_size();
frame->ColorImage.resize(color_size);
memcpy(frame->ColorImage.data(), color_image, color_size);
// Copy the transformation matrix (to master camera)
frame->transform2master = Eigen::Matrix4f(file->transform2master
    );
// Resource recycle
transformed_color_image.reset();
point_cloud_image.reset();
if(images[1].is_valid()) frame->timestamp = images[1].
    get_device_timestamp().count() - (int64_t) file->
    record_config.start_timestamp_offset_usec;
printf("%-32s", file->filename.c_str());
```

```
  for (int i = 0; i < 2; i++)
  {
    if (images[i] != NULL)
    {
      images[i].reset();
      images[i] = NULL;
    }
    else
    {
      printf("--%12s", "");
    }
  }
  return frame;
}
FrameInfo* process_capture(recording_t *file)
{
  k4a::image images[2];
  FrameInfo* frame = new FrameInfo();
  images[0] = file->capture.get_color_image();
  images[1] = file->capture.get_depth_image();
  // Copy depth
  frame->DepthWidth = images[1].get_width_pixels();
  frame->DepthHeight = images[1].get_height_pixels();
  frame->DepthStride = images[1].get_stride_bytes();
  const unsigned depth_size = frame->DepthStride * frame->
     DepthHeight;
  frame->DepthImage.resize(depth_size);
  memcpy(frame->DepthImage.data(), reinterpret_cast<const uint16_t
     *>( images[1].get_buffer()), depth_size * 2);
  k4a::image transformed_color_image = file->k4a_transform.
     color_image_to_depth_camera(images[1], images[0]);
  k4a::image point_cloud_image = file->k4a_transform.
     depth_image_to_point_cloud(images[1],
     K4A_CALIBRATION_TYPE_DEPTH);
  // Copy point cloud
  const size_t cloud_size = point_cloud_image.get_size();
  frame->PointCloudData.reserve(cloud_size * 2);
  frame->PointCloudData.resize(cloud_size);
```

```
  memcpy(frame->PointCloudData.data(), reinterpret_cast<const
      int16_t*>(point_cloud_image.get_buffer() ), cloud_size );
  // Copy color
  frame->ColorWidth = transformed_color_image.get_width_pixels();
  frame->ColorHeight = transformed_color_image.get_height_pixels()
      ;
  frame->ColorStride = transformed_color_image.get_stride_bytes();
  const uint8_t* color_image = \
    reinterpret_cast<const uint8_t*>( transformed_color_image.
        get_buffer());
  const size_t color_size = transformed_color_image.get_size();
  frame->ColorImage.resize(color_size);
  memcpy(frame->ColorImage.data(), color_image, color_size);
  // Copy the transformation matrix (to master camera)
  frame->transform2master = Eigen::Matrix4f(file->transform2master
      );
  // Resource recycle
  transformed_color_image.reset();
  point_cloud_image.reset();
  for (int i = 0; i < 2; i++)
  {
    if (images[i] != NULL)
    {
      frame->timestamp = images[i].get_device_timestamp().count()
          - (int64_t) file->record_config.
          start_timestamp_offset_usec;
      images[i].reset();
      images[i] = NULL;
    }
    else
    {
      printf("--%12s", "");
    }
  }
  return frame;
}
pcl::PointXYZRGBNormal Laser_coord_exract(recording_t *file, const
    int reptime, std::chrono::duration<int, std::micro> laser_time
```

```
 , int color){
Eigen::Vector3f LaserCenter{0, 0, 0};
int frame_rate = 30;
switch(file->record_config.camera_fps){
  case K4A_FRAMES_PER_SECOND_5 :
    frame_rate = 5;break;
  case K4A_FRAMES_PER_SECOND_15 :
    frame_rate = 15;break;
  case K4A_FRAMES_PER_SECOND_30 :
    frame_rate = 30;break;
  default :
    printf("ERROR: Failed to fine frame rate setting!\n");
}
  // Find the nearest frame to the assigned time stamp.
std::cout << laser_time.count() << std::endl;
file->playback.seek_timestamp(laser_time,
   K4A_PLAYBACK_SEEK_DEVICE_TIME);
if (!file->playback.get_next_capture(&file->capture))
{
  printf("ERROR: Recording file is empty: %s\n", file->filename.
     c_str());
}
// Transform depth image to cloud
FrameInfo* frame = process_capture(file);
std::cout << frame->timestamp << std::endl;
if (frame->timestamp - laser_time.count() > 500000/ frame_rate)
   file->playback.get_previous_capture(&file->capture);
pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr source_cloud (new
   pcl::PointCloud<pcl::PointXYZRGBNormal>);
pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr cloud (new pcl::
   PointCloud<pcl::PointXYZRGBNormal>);
pcl::Indices indices;
const int count = frame->ColorWidth * frame->ColorHeight;
const int coord_stride = 1;
const int rep_count = 20;
source_cloud->reserve(count);
std::vector<int> CloudBuf(count * 3);
std::vector<int> ColorBuf(count * 3);
```

```
std::vector<int> valid(count * 3, 0);
std::cout <<"Generating Laser Cloud From Frames, count ="<<
   count << std::endl;
for(int rep_idx = 0; rep_idx < reptime; rep_idx++){
  frame = process_capture(file);
  for (int i_px = 0; i_px < count * 3; i_px++) {
    int PCData_cast = (int) frame->PointCloudData.data()[i_px];
    CloudBuf[i_px] += PCData_cast;
    valid[i_px] += (int) (PCData_cast != 0);
    if((i_px-2) % 3 == 0 && (PCData_cast != 0)){
      int RData_cast = frame->ColorImage.data()[4 * (i_px-2) / 3
          + 2];
      int GData_cast = frame->ColorImage.data()[4 * (i_px-2) / 3
          + 1];
      int BData_cast = frame->ColorImage.data()[4 * (i_px-2) / 3
          + 0];
      ColorBuf[(i_px)] = RData_cast;
      ColorBuf[(i_px + 1)] = GData_cast;
      ColorBuf[(i_px + 2)] = BData_cast;
    }
  }
  file->playback.get_next_capture(&file->capture);
}
for (int i_pt = 0; i_pt < count; i_pt += coord_stride) {
  if (valid[3 * i_pt + 2] == 0) continue;
  if (CloudBuf.data()[3 * i_pt + 2] == 0) continue;
  std::vector<uint8_t> rgbPt{(uint8_t)frame->ColorImage.data()[4
      * i_pt + 2], (uint8_t)frame->ColorImage.data()[4 * i_pt +
     1], (uint8_t)frame->ColorImage.data()[4 * i_pt + 0]};
  std::vector<uint8_t>::iterator id_cmax = std::max_element(
     rgbPt.begin(), rgbPt.end());
  std::vector<uint8_t>::iterator id_cmin = std::min_element(
     rgbPt.begin(), rgbPt.end());
  size_t id_max = std::distance(rgbPt.begin(), id_cmax);
  if (rgbPt[0] == rgbPt[1] && rgbPt[1] == rgbPt[2]) continue;
  int16_t H = 0;
  switch(id_max){
    case 0:
```

```
        H = 60 * ((int16_t)rgbPt[1] - (int16_t)rgbPt[2])/(int16_t)
            ((*id_cmax)-(*id_cmin));
        break;
      case 1:
        H = 60 * ((int16_t)rgbPt[2] - (int16_t)rgbPt[0])/(int16_t)
            ((*id_cmax)-(*id_cmin));
        break;
      case 2:
        H = 60 * ((int16_t)rgbPt[0] - (int16_t)rgbPt[1])/(int16_t)
            ((*id_cmax)-(*id_cmin));
        break;
    }
    if (id_max != color) continue;
    if (color == 0 && (H < -30 && H > 10)) continue; // filter the
        color in interest
    if (color == 1 && (H < -40 && H > 20)) continue; // filter the
        color in interest
    if (color == 2 && (H < -30 && H > 10)) continue; // filter the
        color in interest
    if((*id_cmax) < (char) (100)) continue; // Find the points
       whose brightness (Value) > V/255, valid only when the
       exposure of the camera was squeezed.
    if((*id_cmin) > (char) (0.5 * (*id_cmax)) ) continue; // Find
       the point red enough (S > 50% => (100-S)%)
    // BGR -> RGB
    pcl::PointXYZRGBNormal point;
    point.x = CloudBuf.data()[3 * i_pt + 0] / 1000.0f / (float)
       valid[3 * i_pt + 0];
    point.y = CloudBuf.data()[3 * i_pt + 1] / 1000.0f / (float)
       valid[3 * i_pt + 1];
    point.z = CloudBuf.data()[3 * i_pt + 2] / 1000.0f / (float)
       valid[3 * i_pt + 2];
    point.r = (uint8_t)frame->ColorImage.data()[4 * i_pt + 2];
    point.g = (uint8_t)frame->ColorImage.data()[4 * i_pt + 1];
    point.b = (uint8_t)frame->ColorImage.data()[4 * i_pt + 0];
    if(std::isinf(point.x)||std::isinf(point.y)||std::isinf(point.
       z)) continue;
    if (point.r == 0 && point.g == 0 && point.b == 0) continue;
```

```
    source_cloud->points.push_back(point);
  }
  std::cout << color << "-Color-filtered" << std::endl;
  pcl::StatisticalOutlierRemoval<pcl::PointXYZRGBNormal> sor;
  sor.setInputCloud (source_cloud);
  sor.setMeanK (40);
  sor.setStddevMulThresh (1.0);
  sor.filter (*cloud);
  pcl::removeNaNFromPointCloud(*cloud, *cloud, indices);
  std::cout <<"Constructed-PCL-PointCloud-for-Laser-Marker:-" <<
     std::endl;
  pcl::NormalEstimationOMP<pcl::PointXYZRGBNormal, pcl::
     PointXYZRGBNormal> ne;
  const double normal_radius = 0.01;
  ne.setInputCloud (cloud);
  // Create an empty kdtree representation, and pass it to the
     normal estimation object.
  // Its content will be filled inside the object, based on the
     given input dataset (as no other search surface is given).
  pcl::search::Octree<pcl::PointXYZRGBNormal>::Ptr tree (new pcl::
     search::Octree<pcl::PointXYZRGBNormal>(normal_radius));
  ne.setSearchMethod (tree);
  // Use all neighbors in a sphere of radius 3cm
  ne.setRadiusSearch (normal_radius);
  ne.useSensorOriginAsViewPoint();
  ne.compute (*cloud);
  // Resource recycling;
  file->capture.reset();
  for(auto pt: cloud->points){
    LaserCenter(0) += pt.x;
    LaserCenter(1) += pt.y;
    LaserCenter(2) += pt.z;
  }
  LaserCenter = LaserCenter / (float)cloud->size();
  std::string filename = "../output/Cfilteredcloud_" + std::
     to_string(color) + "Cam" + std::to_string(file->camera_id) +
     ".pcd";
  pcl::io::savePCDFileBinary(filename, *cloud);
```

```
  cloud->clear();
  return pcl::PointXYZRGBNormal(LaserCenter(0), LaserCenter(1),
      LaserCenter(2), (uint8_t)(255*(color == 0)), (uint8_t)(255*(
      color == 1)), (uint8_t)(255*(color == 2)));
}
void CopyIntrinsics(
  const k4a_calibration_camera_t& from,
  CameraIntrinsics& to)
{
  to.Width = from.resolution_width;
  to.Height = from.resolution_height;
  const k4a_calibration_intrinsic_parameters_t& params = from.
      intrinsics.parameters;
  to.cx = params.param.cx;
  to.cy = params.param.cy;
  to.fx = params.param.fx;
  to.fy = params.param.fy;
  to.k[0] = params.param.k1;
  to.k[1] = params.param.k2;
  to.k[2] = params.param.k3;
  to.k[3] = params.param.k4;
  to.k[4] = params.param.k5;
  to.k[5] = params.param.k6;
  to.codx = params.param.codx;
  to.cody = params.param.cody;
  to.p1 = params.param.p1;
  to.p2 = params.param.p2;
}
void CalibrationFromK4a(
  const k4a_calibration_t& from,
  CameraCalibration& to)
{
  CopyIntrinsics(from.depth_camera_calibration, to.Depth);
  CopyIntrinsics(from.color_camera_calibration, to.Color);
  // Extrinsics from depth to color camera
  const k4a_calibration_extrinsics_t* extrinsics = &from.
      extrinsics[K4A_CALIBRATION_TYPE_DEPTH][
      K4A_CALIBRATION_TYPE_COLOR];
```

```
  for (int i = 0; i < 9; ++i) {
    to.RotationFromDepth[i] = extrinsics->rotation[i];
  }
  for (int i = 0; i < 3; ++i) {
    to.TranslationFromDepth[i] = extrinsics->translation[i];
  }
}
void print_body_information(k4abt_body_t body)
{
  std::cout << "Body ID: " << body.id << std::endl;
  FILE* outcsv;
  outcsv = fopen("./body_frames.csv", "a");
  fprintf(outcsv, "New body\n");
  for (int i = 0; i < (int)K4ABT_JOINT_COUNT; i++)
  {
    k4a_float3_t position = body.skeleton.joints[i].position;
    k4a_quaternion_t orientation = body.skeleton.joints[i].
       orientation;
    k4abt_joint_confidence_level_t confidence_level = body.
       skeleton.joints[i].confidence_level;
    fprintf(outcsv, "Joint[%d]: Position[mm] ( %f, %f, %f ); 
       Orientation ( %f, %f, %f, %f); Confidence Level (%d) \n",
      i, position.v[0], position.v[1], position.v[2], orientation.
         v[0], orientation.v[1], orientation.v[2], orientation.v
         [3], confidence_level);
  }
  fprintf(outcsv, "\n\n");
  fclose(outcsv);
}
```

Listing B.8: include/camera_extrinsics.hpp

```
#pragma once
#include <cstdint>
#include <vector>
#include <iostream>
#include <Eigen/Eigen>
#include <k4a/k4a.hpp>
#include <k4arecord/playback.hpp>
```

```
#include <k4abt.hpp>
#include <pcl/io/pcd_io.h>
#include <pcl/common/io.h>
#include <pcl/common/geometry.h>
#include <pcl/common/transforms.h>
#include <pcl/features/integral_image_normal.h>
#include <pcl/filters/passthrough.h>
#include <pcl/filters/extract_indices.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/filters/statistical_outlier_removal.h>
#include <pcl/search/octree.h>
#include <pcl/features/normal_3d_omp.h>
#include <pcl/features/principal_curvatures.h>
#include <pcl/registration/icp.h>
#define BOXCLIP
// #define CHECKJOINTS
// #define BGGENERATE
// Customized structure
struct  CameraExtrinsics
{
  Eigen::Vector3f translation;
  Eigen::Quaternionf rotation;
};
struct CameraIntrinsics
{
  // Sensor resolution
  int32_t Width, Height;
  // Intrinsics
  float cx, cy;
  float fx, fy;
  float k[6];
  float codx, cody;
  float p1, p2;
};
struct CameraCalibration
{
  // Intrinsics for each camera
  CameraIntrinsics Color, Depth;
```

```
  // Extrinsics transform from 3D depth camera point to 3D point
     relative to color camera
  float RotationFromDepth[3*3];
  float TranslationFromDepth[3];
};
struct FrameInfo
{
  // Accelerometer reading for extrinsics calibration
  float Accelerometer[3];
  // Body including the joints, etc.
  Eigen::Matrix4f transform2master = Eigen::Matrix4f::Identity(4,
     4);
  Eigen::Quaternionf rotation2master;
  pcl::PointCloud<pcl::PointXYZRGBNormal> neck;
  k4abt_joint_confidence_level_t confLevel =
     K4ABT_JOINT_CONFIDENCE_NONE;
  // Color image
  std::vector<uint8_t> ColorImage;
  int ColorWidth = 0, ColorHeight = 0, ColorStride = 0;
  // Depth image
  std::vector<uint16_t> DepthImage;
  int DepthWidth = 0, DepthHeight = 0, DepthStride = 0;
  // Body map image
  std::vector<uint8_t> BodyMapImage;
  int BodyMapWidth = 0, BodyMapHeight = 0, BodyMapStride = 0;
  int idx_closest_body = 0;
  // Point cloud data
  std::vector<int16_t> PointCloudData;
  uint32_t CameraIndex;
  int FrameNumber;
  std::string filename;
  uint64_t timestamp = 0;
};
struct AlignmentTransform
{
  float Transform[16];
  bool Identity = true;
  inline void operator=(const Eigen::Matrix4f& src)
```

```
  {
    Identity = src.isIdentity();
    for (int row = 0; row < 4; ++row) {
      for (int col = 0; col < 4; ++col) {
        Transform[row * 4 + col] = src(row, col);
      }
    }
  }
  inline void Set(Eigen::Matrix4f& dest) const
  {
    if (Identity) {
      dest = Eigen::Matrix4f::Identity();
    } else {
      for (int row = 0; row < 4; ++row) {
        for (int col = 0; col < 4; ++col) {
          dest(row, col) = Transform[row * 4 + col];
        }
      }
    }
  }
};
class ExtrinsicsCalibration
{
  public:
    ExtrinsicsCalibration()
    {
    }
    ~ExtrinsicsCalibration()
    {
    }
    bool CalculateExtrinsics(
      const std::vector<FrameInfo*>& frames,
      std::vector<AlignmentTransform>& extrinsics,
      std::vector<uint64_t>& timestamps,
      std::string out_dir);
  private:
    std::vector<pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr>
      full_cloud;
```

```
    bool GenerateFullCloudFromFrames(FrameInfo frame);
    bool GenerateBodyCloudFromFrames(FrameInfo frame);
    bool GenerateBGCloudFromFrames(FrameInfo frame);
    bool DownSample(pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr&
       cloud, const float voxel_size, const int idx);
};
```

Listing B.9: include/LaserCalib.cpp

```
#include <stdio.h>
#include <malloc.h>
#include <iostream>
#include "LaserCalib.hpp"
int main(int argc, char **argv)
{
  if (argc != 4 && argc != 1)
  {
    printf("Usage: ./laser_reg <exp date yyyymmdd> <scenario name>
       <serialnumber>\n");
    return 1;
  }
  size_t file_count = (size_t) 4;
  // Allocate memory to store the state of N recordings.
  recording_t *files = reinterpret_cast<recording_t*>(malloc(
     sizeof(recording_t) * file_count));
  if (files == NULL)
  {
    printf("Failed to allocate memory for playback (%zu bytes)\n",
        sizeof(recording_t) * file_count);
    return 1;
  }
  memset(files, 0, sizeof(recording_t) * file_count);
  std::cout <<"Transforming the images to the point cloud based on
     stored extrinsic at \"build\" folder."<< std::endl;
  std::cout <<"
     ==========================================================="
     << std::endl;
  // Main process to transform the images to the point clouds./
  bool master_found = false;
```

```
k4a_result_t result = K4A_RESULT_SUCCEEDED;
k4a_stream_result_t stream_result = K4A_STREAM_RESULT_SUCCEEDED;
int frame_rate = 30;
if (result != K4A_RESULT_SUCCEEDED) {
  printf("Some error occured in initiation process!");
  return -1;
}
std::string out_dir = ".";
out_dir = "../output/" + std::string(argv[2]) + "/" + std::
   string(argv[3]);
std::filesystem::create_directories(out_dir+"/necks/");
#ifdef CHECKJOINTS
  std::filesystem::create_directories(out_dir+"/joints/");
#endif
std::string dir_Laser_input;
// dir_Laser_input = std::string("../input/") + std::string(argv
   [1]) + std::string("/") + std::string(argv[2]) + std::string
   ("/");
dir_Laser_input = std::string("../../../THzExp/") + std::string(
   argv[1]) + std::string("/") + std::string(argv[2]) + std::
   string("/");
bool terminated = false;
double count = 0;
//Open the video.//
for (size_t i = 0; i < file_count; i++)
{
  files[i].filename = "./"; // For init the string var
  files[i].filename = dir_Laser_input + std::string("SN") + std
     ::string(argv[3]) + std::string("Cam") + std::to_string(i +
      1) + std::string(".mkv");
  files[i].out_dir = out_dir;
  files[i].playback = k4a::playback::open(files[i].filename.
     c_str());
  if (!files[i].playback.is_valid())
  {
    printf("Failed to open file: %s\n", files[i].filename.c_str
       ());
    break;
```

```
    }
    files[i].record_config = files[i].playback.
        get_record_configuration();
    files[i].k4a_calibration = files[i].playback.get_calibration()
        ;
    files[i].camera_id = i;
    files[i].k4a_transform = k4a::transformation(files[i].
        k4a_calibration);
    files[i].playback.set_color_conversion(
        K4A_IMAGE_FORMAT_COLOR_BGRA32);
    if (i == 0) {files[i].record_config.wired_sync_mode =
        K4A_WIRED_SYNC_MODE_MASTER; printf("Opened master recording
        file: %s\n", files[i].filename.c_str());}
    else if (files[i].record_config.wired_sync_mode ==
        K4A_WIRED_SYNC_MODE_SUBORDINATE) printf("Opened subordinate
        recording file: %s\n", files[i].filename.c_str());
    else if (files[i].record_config.wired_sync_mode ==
        K4A_WIRED_SYNC_MODE_STANDALONE) printf("Opened standalone
        recording file: %s\n", files[i].filename.c_str());
    else
    {
      printf("ERROR: Failed to recognize the sync. mode: %s\n",
          files[i].filename.c_str());
      result = K4A_RESULT_FAILED;
      break;
    }
    // Find the nearest frame to the assigned time stamp.
    files[i].playback.seek_timestamp(std::chrono::duration<int,
        std::micro>(1000000), K4A_PLAYBACK_SEEK_DEVICE_TIME);
    if (!files[i].playback.get_next_capture(&files[i].capture))
    {
    printf("ERROR: Recording file is empty: %s\n", files[i].
        filename.c_str());
    }
    FrameInfo* frame = process_capture(&files[i]);
    if (frame->timestamp - std::chrono::duration<int, std::micro
        >(600000).count() > 500000/ frame_rate) files[i].playback.
        get_previous_capture(&files[i].capture);
```

```
}
printf("%-58s %12s\n", "Source file", "TIME");
printf("==========\n");
while (true) {
  count++;
  ExtrinsicsCalibration extrinsicsCalib;
  std::vector<FrameInfo*> frames(file_count);
  std::vector<AlignmentTransform> extrinsics(file_count);
  std::vector<uint64_t>  timestamps(file_count);
  k4a_stream_result_t stream_result;
  // Find the lowest timestamp out of each of the current
     captures.
  for (size_t i = 0; i < file_count; i++)
  {
    if (files[i].playback.get_next_capture(&files[i].capture))
    {
      FrameInfo* frame = process_capture_incbody(&files[i]);
      k4a_imu_sample_t imu_sample;
      printf("%-32s %12ld usec\n", frame->filename.c_str(),
         frame->timestamp);
      files[i].playback.get_next_imu_sample(&imu_sample);
      for (int j = 0; j < 3; ++j) {
        frame->Accelerometer[j] = imu_sample.acc_sample.v[j];
      }
      frame->CameraIndex = i;
      frame->filename = files[i].filename;
      frames[i] = frame;
      timestamps[i]= frame->timestamp;
      extrinsics[i] = frame->transform2master;
      files[i].capture.reset();
      if(frames[i]->DepthHeight * frames[i]->DepthWidth == 0){
        terminated = true;
        break;
      }
    }
    else {
      terminated = true;
      break;
```

```
      }
    }
    if(terminated) break;
    std::cout << "Calculating Extrisincs" << std::endl;
    if (!extrinsicsCalib.CalculateExtrinsics(frames, extrinsics,
       timestamps, out_dir)) {
      std::cout << "Full registration failed" << std::endl;
    }
    for(auto& frm : frames) delete frm;
  }
  std::cout << count << " frames transformed.\nDone\n";
  for (size_t i = 0; i < file_count; i++)
  {
    if (files[i].handle != NULL)
    {
      k4a_playback_close(files[i].handle);
      files[i].handle = NULL;
    }
  }
  free(files);
  return result == K4A_RESULT_SUCCEEDED ? 0 : 1;
}
```

Listing B.10: include/camera_extrinsics.cpp

```
#include "camera_extrinsics.hpp"
#include <fstream>
#include <filesystem>
#include <typeinfo>
namespace fs = std::filesystem;
bool ExtrinsicsCalibration::GenerateFullCloudFromFrames(
  FrameInfo frame)
{
  const int idx = frame.CameraIndex;
  if (!frame.PointCloudData.data()) {
    return false;
  }
  pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr source_cloud (new
     pcl::PointCloud<pcl::PointXYZRGBNormal>);
```

```
pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr cloud (new pcl::
    PointCloud<pcl::PointXYZRGBNormal>);
pcl::Indices indices;
full_cloud[idx] = cloud;
const int count = frame.ColorWidth * frame.ColorHeight;
const int coord_stride = 1;
source_cloud->reserve(count);
for (int i = 0; i < count; i += coord_stride) {
  if (frame.PointCloudData.data()[3 * i + 2] == 0) continue;
  // BGR -> RGB
  pcl::PointXYZRGBNormal point;
  point.x = frame.PointCloudData.data()[3 * i + 0] / 1000.0f;
  point.y = frame.PointCloudData.data()[3 * i + 1] / 1000.0f;
  point.z = frame.PointCloudData.data()[3 * i + 2] / 1000.0f;
  point.r = (uint8_t)frame.ColorImage.data()[4 * i + 2];
  point.g = (uint8_t)frame.ColorImage.data()[4 * i + 1];
  point.b = (uint8_t)frame.ColorImage.data()[4 * i + 0];
  source_cloud->points.push_back(point);
}
pcl::StatisticalOutlierRemoval<pcl::PointXYZRGBNormal> sor;
sor.setInputCloud (source_cloud);
sor.setMeanK (40);
sor.setStddevMulThresh (1.0);
sor.filter (*cloud);
pcl::removeNaNFromPointCloud(*cloud, *cloud, indices);
pcl::NormalEstimationOMP<pcl::PointXYZRGBNormal, pcl::
    PointXYZRGBNormal> ne;
const double normal_radius = 0.01;
ne.setInputCloud (cloud);
// Create an empty kdtree representation, and pass it to the
    normal estimation object.
// Its content will be filled inside the object, based on the
    given input dataset (as no other search surface is given).
pcl::search::Octree<pcl::PointXYZRGBNormal>::Ptr tree (new pcl::
    search::Octree<pcl::PointXYZRGBNormal>(normal_radius));
ne.setSearchMethod (tree);
// Use all neighbors in a sphere of radius 3cm
ne.setRadiusSearch (normal_radius);
```

```
  ne.useSensorOriginAsViewPoint();
  ne.compute (*cloud);
  return true;
}
bool ExtrinsicsCalibration::GenerateBodyCloudFromFrames(
  FrameInfo frame)
{
  const int idx = frame.CameraIndex;
  if (!frame.PointCloudData.data()) {
    return false;
  }
  pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr source_cloud (new
     pcl::PointCloud<pcl::PointXYZRGBNormal>);
  pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr cloud (new pcl::
     PointCloud<pcl::PointXYZRGBNormal>);
  pcl::Indices indices;
  full_cloud[idx] = cloud;
  const int count = frame.ColorWidth * frame.ColorHeight;
  const int coord_stride = 1;
  source_cloud->reserve(count);
  for (int i = 0; i < count; i += coord_stride) {
    if (frame.PointCloudData.data()[3 * i + 2] == 0 || frame.
       BodyMapImage.data()[i] == (u_char) 255) continue;
    // BGR -> RGB
    pcl::PointXYZRGBNormal point;
    point.x = frame.PointCloudData.data()[3 * i + 0] / 1000.0f;
    point.y = frame.PointCloudData.data()[3 * i + 1] / 1000.0f;
    point.z = frame.PointCloudData.data()[3 * i + 2] / 1000.0f;
    point.r = (uint8_t)frame.ColorImage.data()[4 * i + 2];
    point.g = (uint8_t)frame.ColorImage.data()[4 * i + 1];
    point.b = (uint8_t)frame.ColorImage.data()[4 * i + 0];
    if (point.r == 0 && point.g == 0 && point.b == 0) continue;
    source_cloud->points.push_back(point);
  }
  pcl::StatisticalOutlierRemoval<pcl::PointXYZRGBNormal> sor;
  sor.setInputCloud (source_cloud);
  sor.setMeanK (40);
  sor.setStddevMulThresh (1.0);
```

```
  sor.filter (*cloud);
  pcl::removeNaNFromPointCloud(*cloud, *cloud, indices);
  pcl::NormalEstimationOMP<pcl::PointXYZRGBNormal, pcl::
      PointXYZRGBNormal> ne;
  const double normal_radius = 0.01;
  ne.setInputCloud (cloud);
  // Create an empty kdtree representation, and pass it to the
      normal estimation object.
  // Its content will be filled inside the object, based on the
      given input dataset (as no other search surface is given).
  pcl::search::Octree<pcl::PointXYZRGBNormal>::Ptr tree (new pcl::
      search::Octree<pcl::PointXYZRGBNormal>(normal_radius));
  ne.setSearchMethod (tree);
  // Use all neighbors in a sphere of radius 3cm
  ne.setRadiusSearch (normal_radius);
  ne.useSensorOriginAsViewPoint();
  ne.compute (*cloud);
  return true;
}
bool ExtrinsicsCalibration::GenerateBGCloudFromFrames(
  FrameInfo frame)
{
  const int idx = frame.CameraIndex;
  if (!frame.PointCloudData.data()) {
    return false;
  }
  pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr source_cloud (new
      pcl::PointCloud<pcl::PointXYZRGBNormal>);
  pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr cloud (new pcl::
      PointCloud<pcl::PointXYZRGBNormal>);
  pcl::Indices indices;
  full_cloud[idx] = cloud;
  const int count = frame.ColorWidth * frame.ColorHeight;
  const int coord_stride = 1;
  source_cloud->reserve(count);
  for (int i = 0; i < count; i += coord_stride) {
    if (frame.PointCloudData.data()[3 * i + 2] == 0 || frame.
        BodyMapImage.data()[i] != K4ABT_BODY_INDEX_MAP_BACKGROUND
```

```
        || frame.PointCloudData.data()[3 * i + 1] > 900.0f ||frame.
        PointCloudData.data()[3 * i + 1] < -1700.0f) continue;
      // BGR -> RGB
      pcl::PointXYZRGBNormal point;
      point.x = frame.PointCloudData.data()[3 * i + 0] / 1000.0f;
      point.y = frame.PointCloudData.data()[3 * i + 1] / 1000.0f;
      point.z = frame.PointCloudData.data()[3 * i + 2] / 1000.0f;
      point.r = (uint8_t)frame.ColorImage.data()[4 * i + 2];
      point.g = (uint8_t)frame.ColorImage.data()[4 * i + 1];
      point.b = (uint8_t)frame.ColorImage.data()[4 * i + 0];
      if (point.r == 0 && point.g == 0 && point.b == 0) continue;
      source_cloud->points.push_back(point);
    }
    pcl::StatisticalOutlierRemoval<pcl::PointXYZRGBNormal> sor;
    sor.setInputCloud (source_cloud);
    sor.setMeanK (40);
    sor.setStddevMulThresh (1.0);
    sor.filter (*cloud);
    pcl::removeNaNFromPointCloud(*cloud, *cloud, indices);
    return true;
}
bool ExtrinsicsCalibration::DownSample(pcl::PointCloud<pcl::
    PointXYZRGBNormal>::Ptr& cloud, const float voxel_size, const
    int idx)
{
    pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr cloud_filtered(new
        pcl::PointCloud<pcl::PointXYZRGBNormal>);
    pcl::Indices indices;
    // Downsample the point cloud
    pcl::VoxelGrid<pcl::PointXYZRGBNormal> sor;
    sor.setInputCloud (full_cloud[idx]);
    sor.setLeafSize (voxel_size, voxel_size,voxel_size);
    sor.filter(*cloud_filtered);
    cloud = cloud_filtered;
    if (!cloud) {
      std::cout <<"VoxelDownSample failed"<< std::endl;
      return false;
    }
```

```
  // Estimate normals with full resolution point cloud
  pcl::NormalEstimationOMP<pcl::PointXYZRGBNormal, pcl::
    PointXYZRGBNormal> ne;
  const double normal_radius = voxel_size * 5.0;
  ne.setInputCloud (cloud_filtered);
  // Create an empty kdtree representation, and pass it to the
    normal estimation object.
  // Its content will be filled inside the object, based on the
    given input dataset (as no other search surface is given).
  pcl::search::Octree<pcl::PointXYZRGBNormal>::Ptr tree (new pcl::
    search::Octree<pcl::PointXYZRGBNormal>(voxel_size));
  ne.setSearchMethod (tree);
  // Use all neighbors in a sphere of radius 3cm
  ne.setRadiusSearch (normal_radius);
  ne.useSensorOriginAsViewPoint();
  ne.compute (*cloud);
  pcl::removeNaNFromPointCloud(*cloud, *cloud, indices);
  indices.clear();
  pcl::removeNaNNormalsFromPointCloud(*cloud, *cloud, indices);
  return true;
}
bool ExtrinsicsCalibration::CalculateExtrinsics(
  const std::vector<FrameInfo*>& frames,
  std::vector<AlignmentTransform>& output,
  std::vector<uint64_t>& timestamps,
  std::string out_dir)
{
  if (frames.empty()) {
    std::cout <<"No images provided to registration"<< std::endl;
    return false;
  }
  const int camera_count = static_cast<int>( frames.size() );
  output.resize(camera_count);
  full_cloud.resize(camera_count);
  std::vector<Eigen::Matrix4f> current_transform(camera_count);
  std::vector<Eigen::Matrix4f> final_transform(camera_count);
  for (int camera_index = 0; camera_index < camera_count; ++
    camera_index)
```

```
  {
#ifdef BGGENERATE
      GenerateFullCloudFromFrames(*(frames[camera_index]));
#else
    if(frames[camera_index]->BodyMapHeight * frames[camera_index
       ]->BodyMapWidth != 0) {
      GenerateBodyCloudFromFrames(*(frames[camera_index]));
    }else{
      GenerateFullCloudFromFrames(*(frames[camera_index]));
    }
#endif
    std::string matfile = "./Laser_transform" + std::to_string(
       camera_index)+ ".txt";
    std::ifstream fin(matfile);
    Eigen::Matrix4f temp;
    for(int i =0;i<4;i++){
      for(int j =0;j<4;j++){
          fin >> temp(i,j);
      }
    }
    current_transform[camera_index] << temp.cast<float>();
    frames[camera_index]->transform2master << current_transform[
       camera_index].cast<float>();
    fin.close();
  }
  pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr final_cloud(new pcl
     ::PointCloud<pcl::PointXYZRGBNormal>);
  pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr neck_cloud(new pcl
     ::PointCloud<pcl::PointXYZRGBNormal>);
  Eigen::Vector3f neck_est{0, 0, 0};
  int count_valNecks = 0;
   // Save to PLY and JSON after Normal ICP
  for (int camera_index = 0; camera_index < camera_count; ++
     camera_index)
  {
    pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr cloud_i(
       full_cloud[camera_index]);
    pcl::transformPointCloudWithNormals(*cloud_i, *cloud_i, frames
```

```
        [camera_index]->transform2master);
    pcl::transformPointCloudWithNormals(frames[camera_index]->neck
        , frames[camera_index]->neck, frames[camera_index]->
        transform2master);
    if (frames[camera_index]->confLevel ==
        K4ABT_JOINT_CONFIDENCE_MEDIUM)
    {
      count_valNecks++;
      Eigen::Vector3f cur_neck{frames[camera_index]->neck[0].x,
          frames[camera_index]->neck[0].y, frames[camera_index]->
          neck[0].z};
      frames[camera_index]->neck.clear();
      neck_est += cur_neck;
    }
    #ifdef CHECKJOINTS
      pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr cloud_joints(
          new pcl::PointCloud<pcl::PointXYZRGBNormal>);
      std::string filename = out_dir + "/joints/" +std::to_string
          (timestamps[camera_index])+"Cam"+std::to_string(
          camera_index) +".pcd";
      if (pcl::io::loadPCDFile(filename, *cloud_joints)==0)
      {
        pcl::transformPointCloudWithNormals(*cloud_joints, *
            cloud_joints, frames[camera_index]->transform2master);
        pcl::io::savePCDFileBinary(filename, *cloud_joints);
      }
    #endif
    *final_cloud += *cloud_i;
    cloud_i->clear();
  }
#ifdef BOXCLIP
  pcl::PassThrough<pcl::PointXYZRGBNormal> pass;
  pass.setInputCloud (final_cloud);
  pass.setFilterFieldName ("x");
  pass.setFilterLimits (-1.0f, 1.0f);
  pass.filter (*final_cloud);
  pass.setFilterFieldName ("y");
  pass.setFilterLimits (-1.0f, 1.0f);
```

```
  pass.filter (*final_cloud);
  pass.setFilterFieldName ("z");
  pass.setFilterLimits (-1.3f, 0.9f);
  pass.filter (*final_cloud);
#endif
  std::string filename = out_dir + "/"  +std::to_string(timestamps
     [0])+ ".pcd";
  std::cout <<"Generated Cloud, count ="<< final_cloud->size() <<
      std::endl;
  std::cout <<"Saving point cloud"<< std::endl;
  std::cout <<filename<< std::endl;
  pcl::io::savePCDFileBinary(filename, *final_cloud);
  if(count_valNecks) {
    neck_est/=float(count_valNecks);
    std::cout <<count_valNecks << std::endl << neck_est << std::
       endl;
    neck_cloud->push_back(pcl::PointXYZRGBNormal(neck_est[0],
       neck_est[1], neck_est[2]));
    filename = out_dir + "/necks/"  +std::to_string(timestamps[0])
       + ".pcd";
    pcl::io::savePCDFileBinary(filename, *neck_cloud);
  }
  for(auto& cld : full_cloud) {
    cld->clear();
  }
  neck_cloud->clear();
  full_cloud.clear();
  final_cloud->clear();
  return true;
}
```

Bibliography

[1] C. Kang, X. Du, and J. Takada, "Synchronized dynamic channel sounder and posture capture for millimeter wave radio channel suffered from human body shadowing," in *2023 17th European Conference on Antennas and Propagation (EuCAP)*, 2023, pp. 1–5.

[2] ——, "Point cloud-based prediction models of dynamic human body shadowing at 58 ghz," *IEEE Open Journal of Antennas and Propagation*, 2024.

[3] ——, "Point cloud-based diffraction path extraction for dynamic human body shadowing channel at 300 ghz," *IEEE Open Journal of Antennas and Propagation*, 2024.

[4] G. R. MacCartney, S. Deng, S. Sun, and T. S. Rappaport, "Millimeter-wave human blockage at 73 ghz with a simple double knife-edge diffraction model and extension for directional antennas," in *2016 IEEE 84th Vehicular Technology Conference (VTC-Fall)*, 2016, pp. 1–6.

[5] E. Plouhinec and B. Uguen, "Knife-edge diffraction models for human body shadowing prediction," in *2022 IEEE-APS Topical Conference on Antennas and Propagation in Wireless Communications (APWC)*, 2022, pp. 036–041.

[6] U. T. Virk and K. Haneda, "Modeling human blockage at 5g millimeter-wave frequencies," *IEEE Transactions on Antennas and Propagation*, vol. 68, no. 3, pp. 2256–2266, 2020.

[7] X. Chen, L. Tian, P. Tang, and J. Zhang, "Modelling of human body shadowing based on 28 ghz indoor measurement results," in *2016 IEEE 84th Vehicular Technology Conference (VTC-Fall)*, 2016, pp. 1–5.

[8] E. Plouhinec and B. Uguen, "Utd human body models comparison based on dual motion capture and radio measurements," in *2019 IEEE-APS Topical Conference on Antennas and Propagation in Wireless Communications (APWC)*, 2019, pp. 192–197.

[9] S. Mukherjee, G. Skidmore, T. Chawla, A. Bhardwaj, C. Gentile, and J. Senic, "Scalable modeling of human blockage at millimeter-wave: A comparative analysis of knife-edge diffraction, the uniform theory of diffraction, and physical optics against 60 ghz channel measurements," *IEEE Access*, vol. 10, pp. 133 643–133 654, 2022.

[10] K. Bullington, "Radio propagation for vehicular communications," *IEEE Transactions on Vehicular Technology*, vol. 26, no. 4, pp. 295–308, 1977.

[11] ITU-R, "IMT Vision - Framework and overall objectives of the future development of IMT for 2020 and beyond," *Rec. ITU-R M.2083-0*, 2015.

[12] A. Ghosh and M. Kim, "THz Channel Sounding and Modeling Techniques: An Overview," *IEEE Access*, vol. 11, pp. 17 823–17 856, 2023.

[13] V. Petrov, T. Kurner, and I. Hosako, "Ieee 802.15.3d: First standardization efforts for sub-terahertz band communications toward 6g," *IEEE Commun. Mag.*, vol. 58, no. 11, pp. 28–33, 2020.

[14] C. Han, Y. Wang, Y. Li, Y. Chen, N. A. Abbasi, T. Kürner, and A. F. Molisch, "Terahertz wireless channels: A holistic survey on measurement, modeling, and analysis," *IEEE Commun. Surv. Tut.*, vol. 24, no. 3, pp. 1670–1707, 2022.

[15] ITU-R, "Radio Regulations," *Rec. ITU-R M.2083-0*, 2020.

[16] B. Roy, "Anthropometric Notes," 2020. [Online]. Available: https://roymech.org/Useful_Tables/Human/Human_sizes.html

[17] A. Deng, Y. Liu, and D. M. Blough, "Exploring Performance Limits on Proactive Fair Scheduling for mmWave WLANs," in *2022 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, 2022, pp. 1–6.

[18] Y. Oguma, T. Nishio, K. Yamamoto, and M. MORIKURA, "Proactive Handover Based on Human Blockage Prediction Using RGB-D Cameras for mmWave Communications," *IEICE Transactions on Communications*, vol. E99.B, pp. 1734–1744, 08 2016.

[19] A. Bhardwaj, D. Caudill, C. Gentile, J. Chuang, J. Senic, and D. G. Michelson, "Geometrical-empirical channel propagation model for human presence at 60 ghz," *IEEE Access*, vol. 9, pp. 38 467–38 478, 2021.

[20] B. C., *Advanced Engineering Electromagnetics*, 2nd ed. John Wiley & Sons, 2012.

[21] S. D., *Electromagnetic Simulation Using the FDTD Method*. IEEE Press, 2013.

[22] J. M. Eckhardt, C. E. Reinhardt, T. Doeker, E. A. Jorswieck, and T. Kürner, "Capacity analysis for time-variant mimo channel measurements at low thz frequencies," in *2023 17th European Conference on Antennas and Propagation (EuCAP)*, 2023, pp. 1–5.

[23] K. R. of), "Discussion document: The effect of human body blockage on indoor radio wave propagation in the frequency rangeof 47 ghz to 330 ghz," in *ITU-R WP3K/324*, 2023, pp. 1–15.

[24] P. Koivumäki and K. Haneda, "Point cloud ray-launching simulations of indoor multipath channels at 60 ghz," in *2022 IEEE 33rd Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, 2022, pp. 01–07.

[25] K. Saito, N. Keerativoranan, and J. Takada, "Dynamic propagation simulation method from lidar point cloud data for smart office scenario," in *2022 IEEE 33rd Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, 2022, pp. 1–6.

[26] M. Ali and M. Ando, "Computation of slope diffraction by modified edge representation (mer) equivalent edge currents (eecs) line integration," in *2019 URSI Asia-Pacific Radio Science Conference (AP-RASC)*, 2019, pp. 1–4.

[27] T. Murasaki, M. Sato, Y. Inasawa, and M. Ando, "Equivalent edge currents or modified edge representation of flat plates: Fringe wave components," *IEICE Trans. Electron.*, vol. E76-C, no. 9, pp. 1412–1419, 1993.

[28] M. Ali, T. Kohama, and M. Ando, "Modified edge representation (mer) consisting of keller's diffraction coefficients with weighted fringe waves and its localization for evaluation of corner diffraction," *IEEE Transactions on Antennas and Propagation*, vol. 63, no. 7, pp. 3158–3167, 2015.

[29] A. Alyosef, S. Rizou, Z. D. Zaharis, P. I. Lazaridis, A. M. Nor, O. Fratu, S. Halunga, T. V. Yioultsis, and N. V. Kantartzis, "A survey on the effects of human blockage on the performance of mm wave communication systems," in *2022 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom)*, 2022, pp. 249–253.

[30] W. Qi, J. Huang, J. Sun, Y. Tan, C.-X. Wang, and X. Ge, "Measurements and modeling of human blockage effects for multiple millimeter wave bands," in *2017 13th International Wireless Communications and Mobile Computing Conference (IWCMC)*, 2017, pp. 1604–1609.

[31] C. Gustafson and F. Tufvesson, "Characterization of 60 ghz shadowing by human bodies and simple phantoms," in *2012 6th European Conference on Antennas and Propagation (EUCAP)*, 2012, pp. 473–477.

[32] G. A. Thiele and W. L. Stutzman, *Antenna Theory and Design*, 3rd ed. John Wiley & Sons, 2012.

[33] R. Kouyoumjian and P. Pathak, "A uniform geometrical theory of diffraction for an edge in a perfectly conducting surface," *Proceedings of the IEEE*, vol. 62, no. 11, pp. 1448–1461, 1974.

[34] ITU-R, "Propagation by diffraction," *Rec. ITU-R P.526-15*, 2019.

[35] J. Andersen, "Utd multiple-edge transition zone diffraction," *IEEE Transactions on Antennas and Propagation*, vol. 45, no. 7, pp. 1093–1097, 1997.

[36] X. Du and J. Takada, "A uniform additional term using fock-type integral to unify edge diffraction, creeping diffraction, and reflection in lit and shadowed regions," *Prog. Electromagn. Res. B*, vol. 101, no. 6, pp. 101–117, 2023.

[37] P. Pathak, W. Burnside, and R. Marhefka, "A uniform gtd analysis of the diffraction of electromagnetic waves by a smooth convex surface," *IEEE Transactions on Antennas and Propagation*, vol. 28, no. 5, pp. 631–642, 1980.

[38] A. Al-Jzari, J. Huang, and S. Salous, "Mmwave indoor human blockage measurements and modeling at 26, 62, and 70 ghz bands," in *2023 XXXVth General Assembly and Scientific Symposium of the International Union of Radio Science (URSI GASS)*, 2023, pp. 1–5.

[39] G. Chen, J. Takada, K. Saito, M. Kim, and T. Aoyagi, "Development of simultaneous measurement system of mimo channel response and body posture for study of dynamic wban channel," in *55th Annual Conference of Japanese Society for Medical and Biological Engineering*, 2016.

[40] E. Plouhinec and B. Uguen, "A utd elliptic cylinder model for studying body orientation influence on human blockage," in *2023 IEEE-APS Topical Conference on Antennas and Propagation in Wireless Communications (APWC)*, 2023, pp. 068–073.

[41] A. Saito and T. Aoyagi, "Development of a synchronous measurement system for wban channel modeling considering human body motion," in *2022 IEEE 16th International Symposium on Medical Information and Communication Technology (ISMICT)*, 2022, pp. 1–6.

[42] E. Zhang, "Measurement and modeling of the dynamic human shadowing on 30 ghz indoor radio channel," Master's thesis, Tokyo Institute of Technology, 2022.

[43] M. Kim, S. Tang, and K. Kumakura, "Fast double-directional full azimuth sweep channel sounder using low-cost cots beamforming rf transceivers," *IEEE Access*, vol. 9, pp. 80 288–80 299, 2021.

[44] S. IMA., "Evk06002/00," accessed on Oct. 7, 2022. [Online]. Available: https://www.sivers-semiconductors.com/sivers-wireless/evaluation-kits/

[45] E. Olson, "Apriltag: A robust and flexible visual fiducial system," in *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 3400–3407.

[46] T. Song, "Extrinsic calibration for multiple azure kinect cameras," accessed on Oct. 7, 2022. [Online]. Available: https://github.com/stytim/k4a-calibration

[47] OptiTrack, "Calibration squares," accessed on Apr. 16, 2024. [Online]. Available: https://docs.optitrack.com/motive/calibration/calibration-squares

[48] Microsoft, "About azure kinect dk," accessed on Apr. 16, 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/Kinect-dk/about-azure-kinect-dk

[49] M. Jacob, S. Priebe, T. Kürner, M. Peter, M. Wisotzki, R. Felbecker, and W. Keusgen, "Fundamental analyses of 60 ghz human blockage," in *2013 7th European Conference on Antennas and Propagation (EuCAP)*, 2013, pp. 117–121.

[50] J. E. Bjarnason, T. L. J. Chan, A. W. M. Lee, M. A. Celis, and E. R. Brown, "Millimeter-wave, terahertz, and mid-infrared transmissionthrough common clothing," *Applied Physics Letters*, vol. 85, no. 4, pp. 519–521, 07 2004. [Online]. Available: https://doi.org/10.1063/1.1771814

[51] G. Sacco, D. Nikolayev, R. Sauleau, and M. Zhadobov, "Antenna/human body coupling in 5g millimeter-wave bands: Do age and clothing matter?" *IEEE Journal of Microwaves*, vol. 1, no. 2, pp. 593–600, 2021.

[52] J. Eckhardt, A. Schultze, R. Askar, T. Doeker, M. Peter, W. Keusgen, and T. Kürner, "Uniform analysis of multipath components from various scenarios with time-domain channel sounding at 300ghz," *IEEE Open Journal of Antennas and Propagation*, vol. 4, pp. 446–460, 2023.

[53] Microsoft, "Azure kinect body tracking joints," accessed on Apr. 16, 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/kinect-dk/body-joints

[54] J. B. Keller, "Geometrical theory of diffraction∗," *J. Opt. Soc. Am.*, vol. 52, no. 2, pp. 116–130, Feb 1962.

[55] R. Kouyoumjian, "Asymptotic high-frequency methods," *Proceedings of the IEEE*, vol. 53, no. 8, pp. 864–876, 1965.

[56] S. TAKAHASHI and Y. YAMADA, "Propagation-loss prediction using ray tracing with a random-phase technique," *IEICE transactions on fundamentals of electronics, communications and computer sciences*, vol. 81, no. 7, pp. 1445–1451, 07 1998.

[57] S. Collonge, G. Zaharia, and G. Zein, "Influence of the human activity on wide-band characteristics of the 60 ghz indoor radio channel," *IEEE Transactions on Wireless Communications*, vol. 3, no. 6, pp. 2396–2406, 2004.

[58] L. Marple, "Computing the discrete-time "analytic" signal via fft," *IEEE Transactions on Signal Processing*, vol. 47, no. 9, pp. 2600–2603, 1999.

Publications

Journal Papers

1. **CheChia Kang**, Xin Du, and Jun-ichi Takada, "Point cloud-based prediction models of dynamic human body shadowing at 58 GHz," *IEEE Open Journal of Antennas and Propagation [Early access]*, 2024.

2. **CheChia Kang**, Xin Du, and Jun-ichi Takada, "Point Cloud-based Diffraction Path Extraction for Dynamic Human Body Shadowing Channel at 300 GHz," *IEEE Open Journal of Antennas and Propagation, [Submitted]*, 2024.

International Conference (Reviewed)

1. **CheChia Kang**, Xin Du, and Jun-ichi Takada, "Synchronized Dynamic Channel Sounder and Posture Capture for Millimeter Wave Radio Channel Suffered from Human Body Shadowing," in *17th Eur. Conf. Antennas and Propag. (EuCAP 2023)*, Firenze, Italy, Mar. 2023.

2. **CheChia Kang**, Xin Du, and Jun-ichi Takada, "Measurement of the Dynamic Radio Channel at Sub-THz Band Affected by Human Body Shadowing," in *XXXVth URSI General Assembly and Scientific Symposium (URSI GASS 2023)*, Hokkaido, Japan, Aug. 2023.

International Conference (Unreviewed)

1. **CheChia Kang**, Xin Du, and Jun-ichi Takada, "Synchronized Dynamic Channel Sounder and Posture Capture for Millimeter Wave Radio Channel Suffered from Human Body Shadowing," in *4th MC and 4th Technical Meeting of COST CA20120*, Dubrovnik, Croatia, Jan. 2023.

2. **CheChia Kang**, Xin Du, and Jun-ichi Takada, "Synchronized Dynamic Channel Measurement and Motion Capture for Sub-THz Radio Channel Affected by Human Presence," in *5th MC and 5th Technical Meeting of COST INTERACT (CA20120)*, Barcelona, Spain, May 2023.

3. **CheChia Kang**, Xin Du, and Jun-ichi Takada, "Equivelent Edge Currents based Forward Scattering Prediction for Complex Circumference of the Cross-section of Human

Body at 300 GHz Band," in *6th MC and 6th Technical Meeting of COST INTERACT (CA20120)*, Poznan, Poland, Sep. 2023.

4. **CheChia Kang**, Xin Du, and Jun-ichi Takada, "Point Cloud-based Diffraction Path Extraction for Dynamic Human Body Shadowing Channel at 300 GHz Band in Corridor Scenario," in *7th MC and 7th Technical Meeting of COST INTERACT (CA20120)*, Lisbon, Portugal, Jan. 2024.

Contribution Documents for Standardization

1. Jun-ichi Takada, **CheChia Kang**, and Xin Du, "MmWave Indoor Human Blockage Measurements and Modeling at 58 GHz Band," in *ITU-R CG3K6-161*, Online, Apr. 2023.

2. Japan, "Discussion document on dynamic human body shadowing and its prediction at 58 GHz band," in *ITU-R CG3K/324*, Online, May 2023.

3. Japan, "Discussion document on dynamic human body shadowing and its prediction at 300 GHz band," in *ITU-R CG3K21*, Online, May 2024.

Domestic Conference

1. **CheChia Kang**, Enqi Zhang, and Jun-ichi Takada, "Dynamic Shadowing Channel Sounding at 25 GHz band together with Posture Capturing," *IEICE Tech. Rep.*, vol. 121, no. 393, pp. 14-16, Feb. 2022.

2. **CheChia Kang**, Xin Du, and Jun-ichi Takada, "Modeling for Dynamic Radio Channel at Millimeter Wave Band Affected by Human Shadowing," in *IEICE Gen. Conf.*, Saitama, Japan, Mar. 2023.

3. **CheChia Kang**, Xin Du, and Jun-ichi Takada, "Synchronized Dynamic Channel Measurement and Motion Capture for Sub-THz Radio Channel Affected by Human Presence," in *IEICE Soc. Conf.*, Nagoya, Japan, Sep. 2023.