
Implementing full-body pose estimation for Virtual Reality applications

- A brief comparison among attainable tools -

Copyright © Aalborg University 2022

The software used for typesetting the document, and creating figures was Overleaf, Google forms and InVision.



AALBORG UNIVERSITY

STUDENT REPORT

Architecture, Design and Media
Technology

Aalborg University
<http://www.aau.dk>

Title:

Implementing full-body pose estimation for Virtual Reality applications: A brief comparison among attainable tools

Keywords:

Virtual Reality, Full-body tracking, Pose estimation, Computer vision, Kinect, BlazePose

Project Period:

Spring semester 2022

Participant(s):

Maciej Odrowąż-Sypniewski
Sofia Lamda

Supervisor(s):

Cumhur Erkut
Stefania Serafin

Copies: 1**Page Numbers:** 84**Date of Completion:**

May 24, 2022

Abstract:

The use of motion tracking for full body motion in virtual reality enables users to match their physical movement with virtual avatars thereby making the virtual experience more realistic, which offers several benefits at a plethora of applications. Today's full-body tracking systems are too complex (for instance, requiring multiple cameras or high-priced external tools) and must undergo a lengthy calibration process, which contradicts the desire to make VR more portable and integrated. The aim of this project is to create a full body tracking solution that is widely accessible. The paper describes a number of tools used for motion tracking. A comparison of different methods is made, along with a discussion of their outcomes. This leads to the development of an attainable computer vision-based solution.

The content of this report is freely available, but publication (with reference) may only be pursued due to agreement with the author.

Contents

1	Introduction	1
1.1	State of the art applications	2
2	Background and literature review	5
2.1	Virtual Reality and Virtual Environment	5
2.2	Embodied Interaction in Virtual environments	5
2.3	Computer Vision	6
3	Design choices	9
3.1	Choice of tools for the project	9
3.2	Research on HMDs	9
3.3	Research on full body tracking tools for VR applications	10
3.3.1	Computer vision	10
3.3.2	AprilTags	10
3.3.3	Kinect	10
3.3.4	MocapForAll	10
3.3.5	Wearable active trackers	11
3.3.6	Motion capture rig	11
3.4	Full body rigging in Unity	11
4	Implementation	13
4.1	Tools	13
4.1.1	Unity	13
4.1.2	Oculus Quest 2	14
4.1.3	BlazePose	14
4.1.4	Microsoft Kinect	16
4.1.5	Avatar	16
4.1.6	Skybox	17
4.2	Scene Composition	17
4.3	Pose estimation tools implementation	18
4.4	Combining HMD tracking with pose estimation	19

5	Evaluation	21
5.1	Participants	21
5.2	Testing procedure	22
5.2.1	Measurements	23
6	Results	25
6.1	Quantitative Data	25
6.2	Qualitative Data	27
6.2.1	Oral and Written feedback	29
6.2.2	Observed Actions	29
7	Discussion	31
7.1	Methodology	31
7.2	User Feedback	31
8	Conclusion	33
8.1	Future work	33
	Bibliography	35
A	Scripts	39
A.1	VNectModel.cs	39
A.2	PoseVisualizer3D.cs	57
A.3	BodySourceView.cs	62
A.4	FaceManager.cs	73
B	Questionnaire	79

Maciej Odrowąż-Sypniewski
<modrow20@student.aau.dk>

Sofia Lamda
<slamda20@student.aau.dk>

Chapter 1

Introduction

Healthcare, education, and entertainment are just a few fields where virtual reality (VR) is applied. VR experiences utilizing head-mounted displays (HMDs) are becoming increasingly popular as they become more affordable, immersive and user-friendly over time. The most widely used HMDs, however, are only able to offer head, eye, and hand tracking when in a virtual environment, providing a virtual experience that focuses on upper body immersion.

At present, the most common full-body tracking systems are too complicated, as they require multiple cameras, expensive external tools, and lengthy calibration processes, with Microsoft Kinect being one of the most widely used external 3D sensor for VR, mainly due to its affordability. Furthermore, the advancement of computer vision in the last few years has inspired many exciting possibilities, such as movement tracking, using a single monocular RGB camera, in conjunction with neural network algorithms.

On this project, we present a widely accessible solution for full body tracking, without the requirement for any passive or active wearable tracker or markers. The tool works effectively in real time with HMD's (we used the Oculus Quest 2), and cooperates with Virtual Reality devices. Full body tracking was achieved using a Microsoft Kinect V2 device in the first implementation, while the second used the BlazePose 3D estimation model. To allow users to interact with their VR avatars, a minimal virtual environment was developed in Unity platform.

By running a test session, in which users were able to interact with both implementations, we were able to analyze the current project's results. Qualitative and quantitative data were collected throughout an after-testing questionnaire as well as a recording of observed actions and oral feedback was taken, in order to assess the Kinect V2 and BlazePose implementations and compare them. The contributions of this project include:

- A thorough review of the existing approaches and tools for full body tracking.

- A VR application that includes two different methods of body tracking, one using Microsoft Kinect V2 and the other using BlazePose. Presenting and analyzing the results of both implementations, as well as a comparison between them.
- A result presentation of both implementations as well as a comparative analysis.

1.1 State of the art applications

HybridTrak

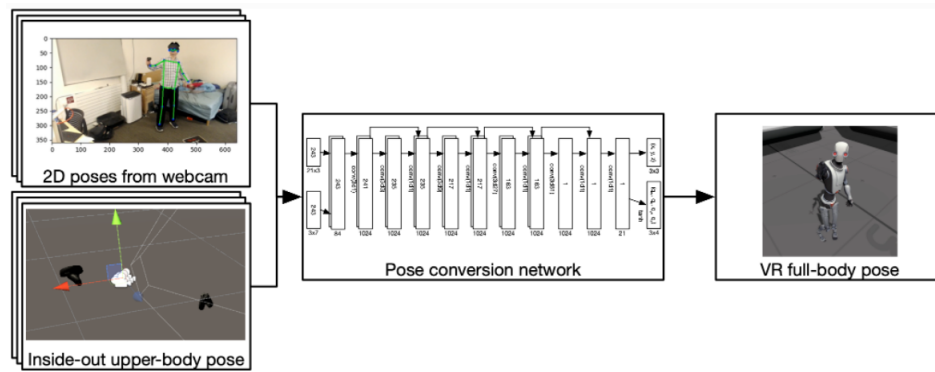


Figure 1.1: HybridTrak: Full-Body Tracking to VR tool [35].

HybridTrak (fig.1.1) is a real time full-body tracking tool introduced by Stanford University in 2022. HybridTrak uses a single uncalibrated RGB camera for accurate results instead of complex external tools such as RGBD cameras, converting 2D upper body poses from RGB cameras into 3D poses by leveraging poses' tracking data. A neural network processes the converted data to produce the lower body movement[35].

KOS

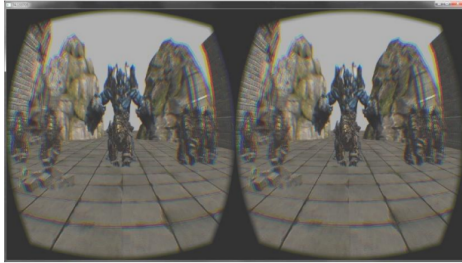


Figure 1.2: KOS: VR game using Kinect V2 and Oculus Rift [12].

Andong National University released KOS (fig.1.2) in 2016. In order to increase efficiency, the game uses three different input devices: Kinect, Oculus Rift, and a smartphone. Besides Kinect for tracking hands, the smartphone is also utilized for swiping, touching, and dragging. KOS was developed in three parts: the client side, which implements the virtual reality game, the smartphone part, which is the input device, and the server, that hosts all

client-phone communication [12].

DUKE

Duke (fig.1.3) is a VR first person shooter (FPS) developed in 2016. It combines technologies such as Oculus Rift, Microsoft Kinect, and Leap Motion. The result of this project is a fully immersive prototype is , which involves foot and torso movement and provides navigation and combat interactions. Duke detects multiple joints of the body, including the hands, head, body, and feet, so that natural body movements can be captured. Kinect sensors provide foot and spine data, while Leap motion sensors provide hand recognition. The VR image can also be generated by tracking the user's head position using the Oculus Rift. Game development was done with Unity Engine[1].

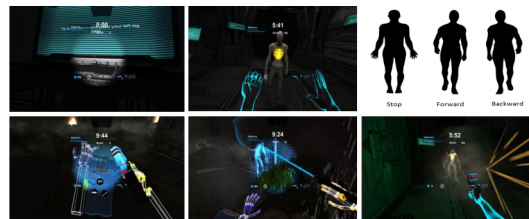


Figure 1.3: DUKE: Enhancing Virtual Reality based FPS Game with Full-body Interactions [1].

ExerGame



Figure 1.4: ExerGame: Promoting Physical Activity [26].

[26].

Metaspace

Metaspace (fig.1.5) is a full body tracking multiperson tool introduced by MIT Media Lab in 2015. Users can experience VR worlds together when they are co-located. Participants own an avatar controlled by their body movements, thus they can see and interact with each other in the Virtual Reality world. Two major components make up the Metaspace tool. The first one contains two Oculus Rift HMDs simultaneously operating in a VR environment. In the second part, two Kinect devices are used to recognize each individual's full body. A four level client - server architecture is used to allow simultaneous co-existence on the same VR space[25].

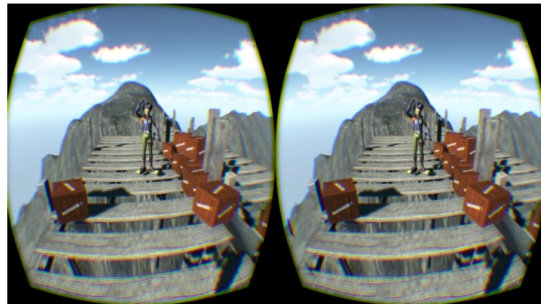


Figure 1.5: MetaSpace: a VR immersive full-body Tracking application [25].

ExerGame (fig.1.4), presented in 2022, is a VR application that aims to prevent WMSDs (work-related musculoskeletal disorders). While in the virtual space, users are able to control their full body avatar. Through this kind of immersive experience, users will be motivated and encouraged to increase their physical activity levels. Using the Oculus Rift VR headset and Microsoft Kinect depth sensor, the full-body VR experience was achieved. Unity Engine was used to develop the game application

Chapter 2

Background and literature review

Chapter two presents an explanation and evaluation of the concept of virtual reality and virtual environments, embodied interaction in virtual environments as well as computer vision.

2.1 Virtual Reality and Virtual Environment

In 1989, Lanier proposed the term "virtual reality" to describe computer simulation, defining the term as "the idea that combines virtual worlds with networking, placing multiple participants in a virtual space using head-mounted displays"[11]. "A computer system capable of creating a virtual world in which a user can immerse himself, roam around, and operate objects" is referred to as virtual reality by the IEEE standard protocol submitted by the work group on virtual reality. The first virtual reality applications were in entertainment and simulation training. Today, virtual reality projects are used in areas such as aeronautical research, architecture, and scientific visualization in defense, medicine, and education[38].

Computer-generated virtual environments can describe objects within the simulation as well as the rules and relationships that govern them. A virtual reality system creates a virtual presence by displaying objects in virtual environments and allowing interaction. The content of the virtual environment such as objects and character determines the virtual environment, while user perception of this content may involve sight, hearing, and touch[17].

2.2 Embodied Interaction in Virtual environments

As Dourish defined in 2001 "embodied interaction" refers to interactions with technology that inhabits our "real world," one that is physically, socially, and geographically situated, and in turn exploits this inhabitation as it interacts with us.

According to Dourish, embodied interaction refers to creating meaning through engaging with artifacts and manipulating it, through the introduction of a phenomenological framework[18].

Through the use of virtual self-avatars, virtual reality can offer a feeling of embodiment. Immersive virtual reality experiences are character-based experiences in which you represent yourself using your own avatar, acquiring self-location, body ownership, and global motor control. When a mismatch occurs between the virtual and the physical body in virtual worlds, the experience of immersion is shown to cause inter-sensory conflict. As well as reducing implicit racial bias, immersion in virtual worlds has been shown to affect negative interpersonal attitudes. According to Lobera et al., "a full body ownership illusion in VR leads to the unification of the virtual and real bodies into one entity"[20].

As is indicated by Taylor, a key feature of avatars is embodiment, i.e., giving participants a virtual body to perform daily body practices. It includes physical acts such as sitting, gesturing, smiling, and touching objects and people as well as social ones, for example appointing friends and dressing in a way that is in line with the hierarchy of power in a community. By using these bodily practices, avatars expand the range of possibilities for communication beyond explicit, textual interaction, thus recapturing some of the body's capacity for non-conceptual interaction[24]. As Becker and Mark stated, the way the user constructs their avatar with regard to appearance, personality and behavior is embedded in a system of meaning informed by the social norms and conventions shaped by both the actual and the virtual world. Nowak and Rauh highlight some of the social norms of avatar appearance. In a study of people's perceptions of avatars, they found that the more anthropomorphic and gendered avatars looked, the more attractive, credible and homophilous they were judged to be[24].

2.3 Computer Vision

As a branch of Artificial Intelligence, Computer Vision forms the basis for the automatic identification and analysis of parameters that provide useful information with regard to an object or a scene with the implementation of deep learning models. A computer vision application has at least one of the following characteristics, as articulated by Thomas S. Huang at the 1996 International Conference of Pattern Recognition[33]: (1) By combining human and machine intelligence, maximum performance can be achieved. (2) The application allows for some mistake flexibility. Human beings play a crucial role in computer vision research. As a result, some errors may occur. (3) Supplementary to vision, other modalities that can also be used[9].

In its simplest form, a computer vision system can be comprised of a high-resolution camera and a computer. An input device, such as a camera, receives

input (frames) from the user. These frames are then processed by the interpreting device (computer) before being analyzed. The next step involves analyzing the input to find distinguishing information, a process called feature extraction. Using the features extracted on the previous step, the final model predicts and classifies the object based on its features[39].

The development of real time computer vision applications has been one of the greatest challenges this scientific field has been facing over the years. Computer vision applications that need to process real-time data are most likely to suffer from a combination of factors, primarily the complicated definition of the human body, and the speed at which the body moves[39]. There can be so many factors that interfere with the viewing of a user's body, including the intensity of the light and the simplicity or not of the background. It is therefore necessary to bring together a number of scientific study areas, such as artificial intelligence, image processing, pattern recognition and machine learning, in order to succeed[10].

Chapter 3

Design choices

Chapter 3 explores the options for tracking the entire body and how it can be accomplished. Furthermore, more detail will be provided on the functions and features of the applications.

3.1 Choice of tools for the project

Thanks to recent advances in image analysis, particularly in computer vision technology, the field of motion tracking has become more and more accessible. Currently, tracking the entire human body can be achieved using no more than a single RGB monocular camera and a medium-performance PC. Before this technology became accessible, tracking was achieved using either an array of cameras and markers or by mounting a series of active trackers on the object.

On this project we have decided to aim for solutions that are most widely accessible, especially in the terms of their costs. We also focused on solutions that did not require any passive or active wearable trackers or markers to make it work.

3.2 Research on HMDs

The solution created during the completion of this project is intended to work with any Head Mounted Display used for Virtual Reality. There is range of HMDs currently available on the market among them products from HTC and Oculus such as HTC Vive and Oculus Rift S. During the project development we have used Oculus Quest 2. Even though it is capable of running VR on its own - without the need for connecting it to a computer - we used it as a wearable display with the intention of creating a PC VR type of application.

3.3 Research on full body tracking tools for VR applications

Below is a list of different types of tools that could be used to achieve full-body tracking that were considered during the completion of this project. It is being discussed which techniques were chosen and why.

3.3.1 Computer vision

In this group we have gathered all solutions that do not require the user to wear any additional devices or markers and are based solely on computer vision. The core of all those techniques is oriented around Machine Learning. It uses a pre-trained neural network that analyses pictures from the camera in real-time and returns a coordinate map of all visible human joints. Since we aimed at using an open source technology, our possible solutions include: OpenPose (2016)[4], Tensorflow's PoseNet (2017)[30], MoveNet (2021)[31] and MediaPipe's BlazePose (2020)[2] amongst many. Since the project aims towards testing a technology that could be used in Virtual Reality, we had to exclude options that return a two dimensional joints map and opt for those offering three dimensional maps. This requirement narrowed our options down to OpenPose and BlazePose. We decided to go with BlazePose because OpenPose did not feature a native support for single RGB monocular camera.

3.3.2 AprilTags

The second category contained techniques where tracking was achieved using cameras that would track fiducial markers - AprilTags. Since one of our aims was to implement a markerless solution we have decided not to include this in the final assessment.

3.3.3 Kinect

Microsoft has released a line of motion controllers called Kinect. They were initially released as peripherals for Microsoft's Xbox game console[32]. Kinect is a motion tracking solution that does not require any handheld trackers. A Kinect based approach to full body tracking appeared promising to us since it was based on a different technique (depth sensing based) unlike other considered solutions and its price fulfilled our requirement for accessibility.

3.3.4 MocapForAll

Another open source motion capture technique based on computer vision but working with pictures from multiple monocular cameras we came across in our

research was MocapForAll[8]. It combines multiple two dimensional maps created using MoveNet into a singular three dimensional map of joints.

3.3.5 Wearable active trackers

The fifth category contains tools that work with active trackers that are attached to the user's body. One of examples for this approach could be achieved using a combination of Vive HMD with two controllers and three more trackers. Those trackers would be attached to the user's waist and feet. We have eliminated this solution from our pool because of its high price.

3.3.6 Motion capture rig

The last group covers all industry grade tools used for motion capture. Those tools are used in e.g. film production and game development[16]. An example would be a Rokoko Smartsuit or system based on multiple OptiTrack brand cameras. Those tools offer high precision full body motion capture, but the price range of those products was out of the project scope.

3.4 Full body rigging in Unity

A significant part of this project is the solution used for full body rigging. The project implements different pose estimation tools that return data in the form of 3D coordinates of each of the user's joints in real time. This data is then being used to control the position of the avatar's joints in Unity. The position of the avatar's joints reflects the position of user's joints. The project required a rigging solution, a solution for controlling the avatar's limbs movements. In character animation, movement is achieved by rotating objects to predetermined angle values. The position of each child joint is calculated according to the position of its parent joint, a technique called forward kinematics. When the reference system is inverted and the position of the child joints is defined by their absolute position rather than by their position subjective to the parent, it is called inverse kinematics[13].

After a thorough research we came to the conclusion that avatar rigging could be done using two different approaches. One of them can be achieved using inverse kinematics based solution built into Unity. This would require to create a dependency chain between each adjacent limb in the avatar prefab in the hierarchy structure of the Unity's project scene. Such process would be time-consuming and it would require to be redone every time the avatar model was changed by the user of the application.

The second option used a different approach; it was almost entirely code based[21]. It referenced each of the avatar joints by their names making use of the Avatar

Configuration tool build into Unity. This solution allowed for much quicker configuration since an avatar following standard naming convention for joints could be automatically mapped by the tool. This approach also speeds up the workflow when the user wants to switch the avatar model.

Each utilized pose estimation tool returns the joints coordinates data in a different format. The positions of joints also differ between the tools. In order to compensate for those differences the script containing the rigging solution behaves differently depending on which pose estimation tool is selected.

Chapter 4

Implementation

This chapter describes how the application works. Details of how the design procedure of the project was conducted and how the user interface and user experience were taken into account will be addressed in more detail.

4.1 Tools

In this section the tools used in the project are described

4.1.1 Unity



Figure 4.1: Unity platform logo[29].

Unity3D (fig.4.1) is a real time development environment for creating games, interactive media and applications. David Helgason, Unity CEO stated that “it is a toolset used to build games, and it’s the technology that executes the graphics, the audio, the physics, the interactions and the networking.”. Unity’s first ever version was created by Joachim Ante, Nicholas Francis and David Helgason, and released in 2005 in Denmark. The first and main goal of the platform was to create an affordable game engine with professional tools for all levels of developers[6]. Unity3D supports platforms such as iOS, iOS Pro, Android, Android Pro and Asset Server. Interesting features, like internal encapsulated platform-related operations,

provide an easy to use programming experience. Unity3d is famous for the good cross platform ability that offers among its three main scripting languages: C#, Javascript and Boo. As a result, application can be deployed on different platforms including Windows, Mac, Xbox 360, PlayStation 3, Wii, iPad, iPhone and Android[34].

4.1.2 Oculus Quest 2



Figure 4.2: The Oculus Quest 2 Headset [28].

Oculus Quest 2 (fig.4.2) is a Virtual Reality headset developed by Reality Labs, officially presented on September 16, 2020. The Quest 2 is shipped with an Android based operating system that allows it be used as a standalone headset, on top of its ability to run on a desktop when connected over Wi-Fi or usb port, through an Oculus compatible VR software. Using the Qualcomm Snapdragon XR2 Platform, Quest 2 can provide higher AI capability. With regard to displaying capabilities, 1832 x 1920 pixels per eye and 50% more pixels than the original Quest, offer a high end visual experience[3].

4.1.3 BlazePose



Figure 4.3: Examples of BlazePose tracking [23]

BlazePose (fig.4.3) is a real time convolutional neural network architecture for human pose estimation introduced by Google research in 2020. BlazePose can be used as a valuable tool in a plethora of applications such as sign language recognition, gestural control and health tracking. BlazePose offers an innovative pose tracking tool and a lightweight body pose estimation neural network. While running, 33 main body key points (fig.4.4) 3D coordinates are being produced with the speed of 30 frames per second.

The tool consists of a real time performance tracker - detector, responsible for tasks like face and hand landmark prediction. First the tracker predicts 33 key point coordinates, the human presence and the region of interest. If no human presence is detected the whole set up reruns on the next frame. When it comes to its neural network architecture, the result is the outcome of offset, heatmap and regression approach. During the training stage, heatmaps and offset data are used. Right after embedding supervision, in which heatmap data are also involved, comes the utilisation of the network by the regression encoder network[2].

On this project BlazePose is implemented through the BlazePose Barracuda Unity package which runs the BlazePose pipeline in the Unity projects[22].

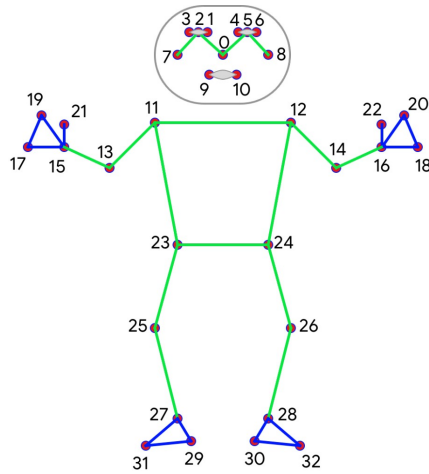


Figure 4.4: BlazePose landmark [23]

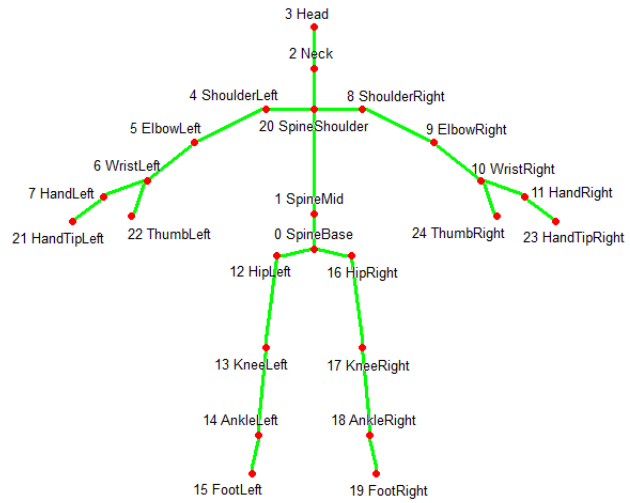


Figure 4.5: Kinect landmark [19].

4.1.4 Microsoft Kinect



Figure 4.6: Microsoft Kinect V2 [14].

Microsoft Kinect (fig.4.6) is a motion sensing input device, originally developed for Xbox video game consoles, introduced in 2010 by Microsoft. Kinect V2 contains a certain number of sensing hardware: a depth sensor, an RGB camera and a four microphone array, offering a full body capture, facial, gesture and voice recognition capabilities[37]. The RGB Kinect camera provides 640×480 pixels, operating at 30 Hz, with the ability to produce 1280×1024 pixels while running at 10 frames/s. With regard to the depth sensor, Kinect incorporates an IR projector and an IR camera, ranging from 0.8 to 3.5 meters distance, producing 640×480 pixels, with a 30 frames/s

rate[7].

In Kinect V2 depth maps are created using a method that measures the time that the light needs to bounce back from a reflected object. This, combined with the constant variable of light speed, can calculate the final distance from various objects. One of the most significant characteristics Kinect V2 has, is the real time automatic identification of anatomical human landmarks (fig.4.5), while using the most common depth camera and a user friendly SDK, which is what makes it the most popular method when it comes to kinematics calculations, spatiotemporal movement and skeleton tracking. According to the Microsoft research team the skeleton tracking is based on training a randomised decision forest algorithm, implementing 100.000 different movement data samples[5].

4.1.5 Avatar

A humanoid avatar model was chosen for the project. It was intended that the avatar would be neutral so that its features would not distract users during the testing procedure. Having also computer's performance in mind a lowpoly skeleton model was chosen[27].

4.1.6 Skybox

A simple skybox depicting a blue sky with clouds was implemented as a scene background in place of the Unity's default skybox. The picture used is actually a rendering of artificially made clouds but it fits the chosen lowpoly avatar model[36].

4.2 Scene Composition

The virtual scene is consisted of a 10x10m square monochromatic floor and a 6x3m preview screen that is positioned on the edge of the floor square. The screen could either display live preview from the webcam or behave like a virtual mirror (fig.4.7) (fig.4.8). The virtual mirror was used to help users in seeing their entire avatar while performing movement routine during the test procedure (fig.4.9). The rest of the scene was empty except of the visible skybox in the background.

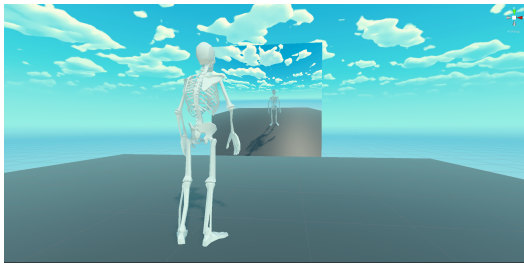


Figure 4.7: Idle position: Scene View

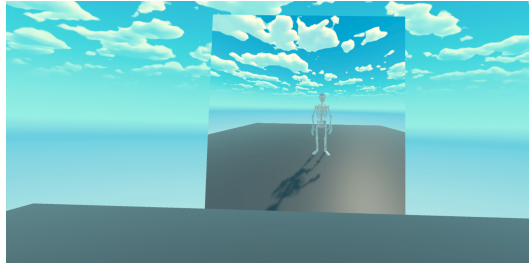


Figure 4.8: Idle position: First person view

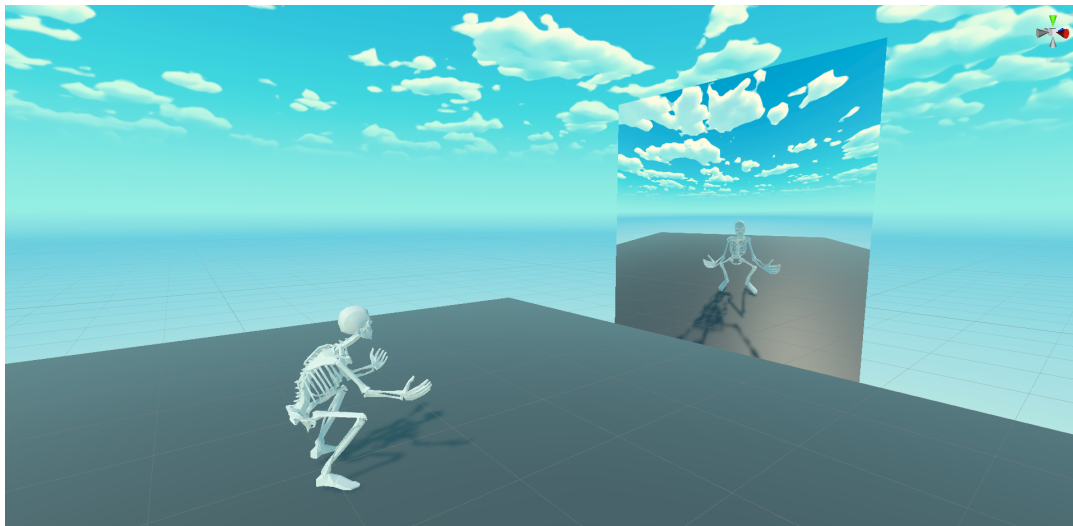


Figure 4.9: Squating position: Scene view

4.3 Pose estimation tools implementation

Two different methods for full body tracking were implemented. First one of them was achieved using BlazePose algorithm (fig.4.10) that was fed a live video feed from a webcam. The second solution was based on the Kinect V2 and the Kinect for Windows SDK 2.0 (fig.4.11)[15]. Even though each implementation used different methods to assess the user's joints positions most part of the code that was used to control the avatar was common for the two versions.

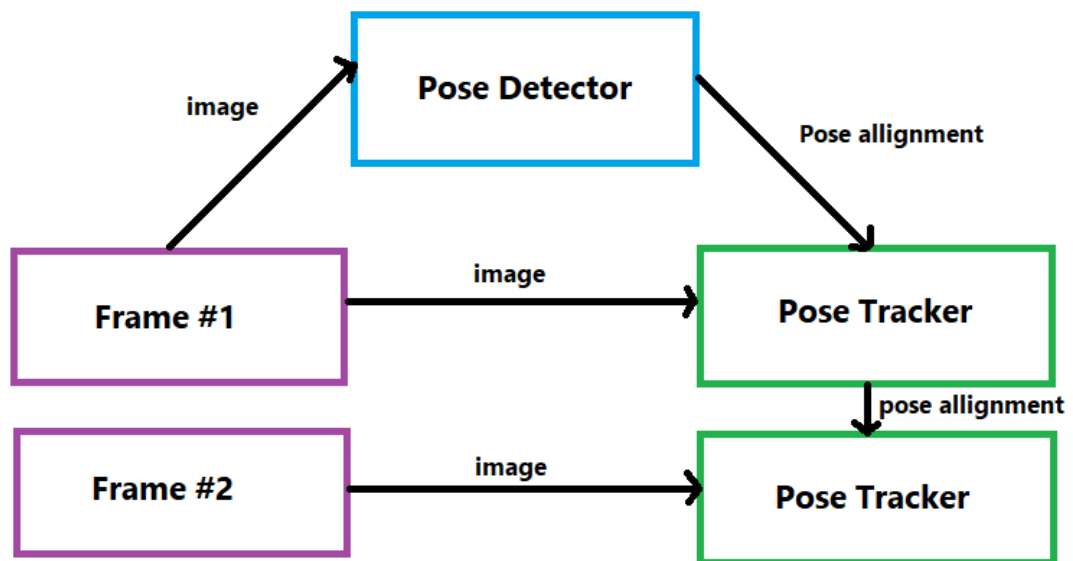


Figure 4.10: BlazePose's human pose estimation pipeline overview.

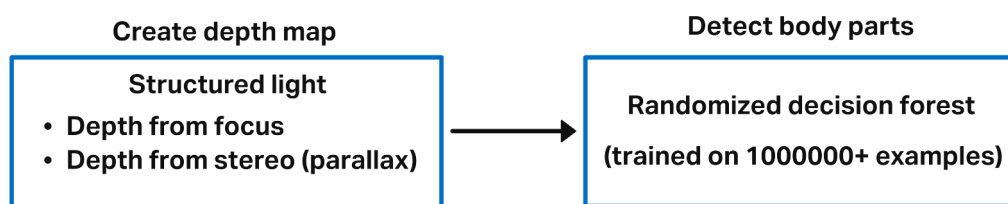


Figure 4.11: Kinect's body position detection pipeline.

4.4 Combining HMD tracking with pose estimation

A significant part of the project was to create a way for combining the tracking of user's head and hands coming in from HMD and controllers with the whole body joints maps coming in from each of the pose estimation tools (fig.4.12). Each of the tools: BlazePose, Kinect V2 and HMD returns coordinates in a different three dimensional reference system. In order to make the avatar work it was crucial to align orientations and compensate for eventual scaling differences between each of those systems. HMD coordinate system is used as the main reference to which the individuals are being adjusted to. A calibration routine has been implemented to compensate for different heights and arms span of different users using the application. During the calibration each user is asked to stand in T-pose and the scaling multiplier is being calculated based on the difference between the corresponding joint coordinates resulting from each of the references systems.

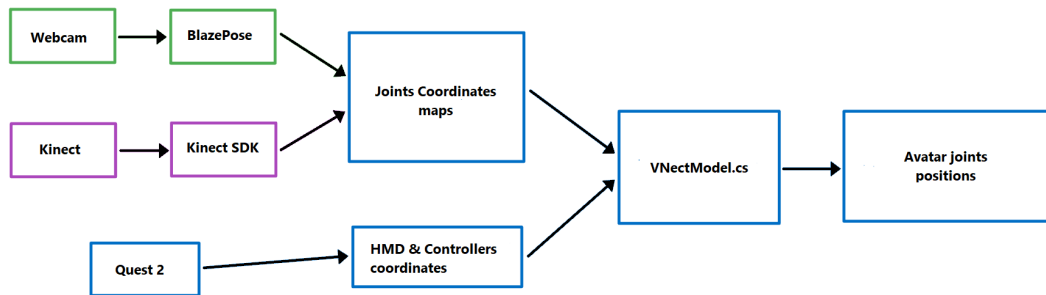


Figure 4.12: Tool's main pipeline overview

Avatar's head position is estimated using HMD tracking, hand controllers, elbows, chest, spine, hips and legs - using results retrieved from pose estimation tools. Because the response of BlazePose estimation was slightly delayed compared to the HMD tracking an additional fix was added. The position of the avatar's elbows is calculated using both elbows coordinates from BlazePose algorithm as well as the positions of the controllers.

Chapter 5

Evaluation

The present research aims to:

- Create affordable full body estimation tools working with Oculus Quest 2
- Test and compare the level of immersiveness between the different methods of implementation.
- To test and compare a number of characteristics and constraints such as speed, lagging, 360 turning and accuracy of movements in each direction.

To evaluate the outcome of this project, we have used a number of tests to examine the quality of the developed tools, in terms of immersiveness, user interaction, the overall experience as well as a comparison between the two methods. Testers were encouraged to use the think-aloud method, and have been recorded during the testing session, providing an abundant amount of qualitative data. Quantitative data was collected through online questionnaires, which users were asked to fill in right after their testing session.

5.1 Participants

The research testing was conducted at Aalborg University in Copenhagen, Denmark, with 20 subjects ages 18-34. No specific criteria were used to select the participants for this study, since the aim is to reach people from diverse backgrounds and ages.

5.2 Testing procedure



Figure 5.1: Warm up routine testing

The pose estimation tools were tested physically, while the questionnaire was filled in on Google Forms. Participants were testing the application individually. In addition, they were informed about the idea behind the project and the enquiries of the research.

Five short videos were shown to each participant before the testing procedure. Each of them showcased a different movement:

- Walk in place (30 seconds): This exercise aims to test how the tools react on an above-average speed body movements.
- Neck rolls (30 seconds): head movement and rotation is being checked.
- Regular and side lunges (30 seconds): While users are doing lunges, depth movement against the side movement is being tested.
- Limbs circulations (30 seconds): At this point, we test the way each one of the tools behave.
- Squats (30 seconds): On the last exercise, an overall full body movement is being tested with the focus on vertical movements.

After providing instructions regarding the warm up routine, users were asked to wear the Oculus Quest 2 and run the test twice (Figs. 5.1 and 5.2), one time for each implemented pose estimation method. Half of the participant tested Kinect-based solution first, before BlazePose. The second half of the group tested the implementations in the opposite order.

On the last stage of testing, the users were asked to fill in an online questionnaire, providing answers about the overall experience of each method, and compare between the two.



Figure 5.2: Calibration position

The testing procedure was designed to last no longer than 12 minutes for each user, including both the testing and the questionnaire, while 2-5 minutes were used in order to introduce the whole testing process to the testers.

5.2.1 Measurements

An online questionnaire, recordings, and the think-aloud method were used to collect the Qualitative and Quantitative data of the aforementioned evaluation process.

There were three different sections on the online questionnaire: Kinect implementation, BlazePose implementation and a comparison between them. Testers were asked to answer, choosing between an 1-7 point scale and open-ended questions.

On the first section there were questions concerning the Kinect implementation method, as shown below:

- Do you believe that the avatar was accurately representing your body movements? If not, which parts of your body you feel were less accurately represented.
- Did you notice any latency in the avatar's movement while you were performing the warm-up routine? If yes, which avatar body parts did you feel were most affected?
- Where there any body parts that were glitching during the test? If yes, which parts of the avatar's body were affected.
- Which movement direction was represented accurately?
- Did you feel that movements of both upper and lower part of your body were represented accurately by the avatar?
- How did you perceive the overall experience in sense of being in the virtual environment?

On the second section, users were asked to answer to the exact same questions with respect to the BlazePose implementation.

On the last section of the questionnaire, participants were asked to answer a number of questions concerning the comparison between the two implementation methods, listed below:

- In your opinion, which implementation was more accurately representing your body movements?

- In your opinion, which implementation had less latency in the avatar's movement while you were performing the warm-up routine?
- In your opinion, which implementation had less glitching incidents in the avatar's movement?
- In your opinion, which one of the implementations felt more realistic, if any?
- If neither felt realistic, please note why.
- Any additional comments on the overall experience?

Chapter 6

Results

Presented in the following section are the results gathered from user experiences. As well as quantitative data collected from the open-ended questions in the questionnaire, qualitative data was acquired from the mini-interviews and from observing the subjects.

6.1 Quantitative Data

Self-report measurement analysis consists of four sections, concerning general information, the Kinect implementation, the BlazePose implementation, and the comparison between them.

User gender and age were asked in the first section.

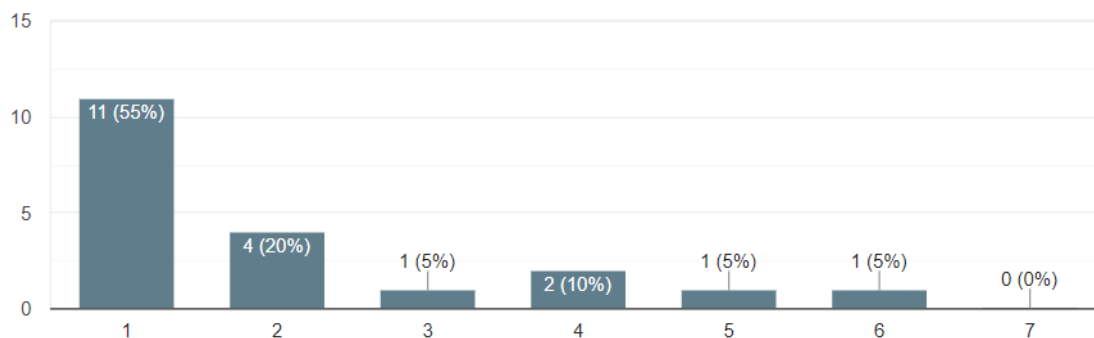


Figure 6.1: Speed accuracy comparison between Kinect (1-3) and BlazePose (5-7)

The participants were asked in section two of the questionnaire to rate their overall experience, on a 7-point scale from 1 (strongly disagree) to 7 (strongly agree), according to their agreement or disagreement with it. The majority of

participants (80%, with 45% pitching the scale extreme), felt that their body is accurately portrayed by their avatar, with the lower body being the part that is less accurately represented. Most of the people (80%, including 45% who selected the scale extreme) stated that they did not experience any latency in the avatar’s movement, with the lower body being the part that shows latency in some cases. There were some glitches on the lower part of the body observed by 55% of respondents (with 15% picking scale extreme). In terms of movement representation, 75% agreed that right and left movement was accurately portrayed, 80% agreed that up and down movement was correctly portrayed, and 55% said that back and forth movement was accurately portrayed. Comparing the upper and lower body (fig.6.5), 60% said the upper body was represented more accurately, while 35% said the two parts were equally represented. Finally, according to the results of the survey, 70% of the participants felt their movement was realistic while on the virtual environment, with 10% reaching the scale extreme.

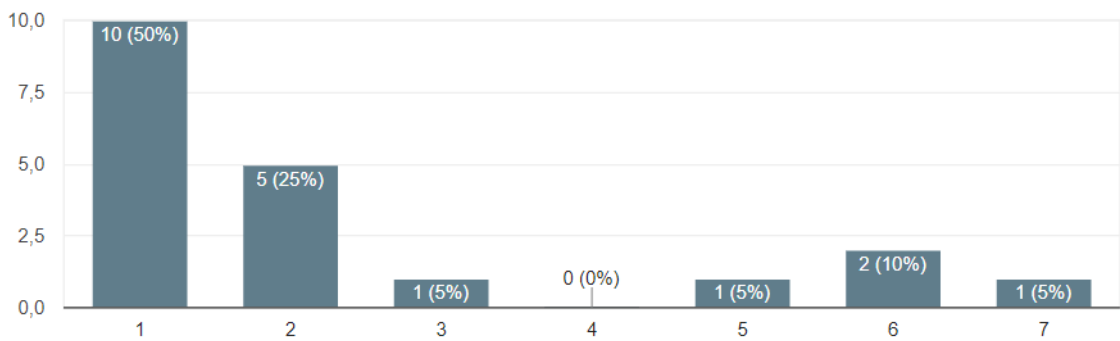


Figure 6.2: Accuracy comparison between Kinect (1-3) and BlazePose (5-7)

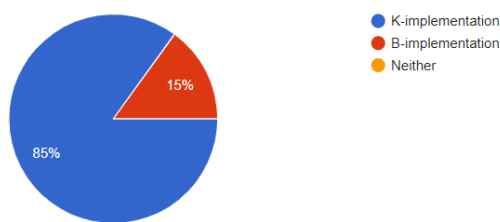


Figure 6.3: Realistic movement comparison between Kinect and BlazePose

Participants were asked to rate on a scale from 1 (strongly disagree) to 7 (strongly agree) their opinion on BlazePose implementation in the third part of the questionnaire. Overall, 55% (none of which selected the scale extreme) of the participants said that their body is accurately represented by the avatar, followed by a unanimous response that the lower part felt less accurately represented. Participants were

divided regarding their perception of latency when it came to noticing it; 50% were not aware of any, and 50% were aware of it (with 20% picking the scale extremes) and the lower body being the most affected part by this latency. Most of them

(85% - 10% choosing scale extreme) noticed several glitches on the lower part of the avatar's body as well as the elbows. According to 73.3% of the participants, right and left movements were accurately portrayed, 60% said up and down movements were accurately portrayed, and 46.7% said back and forth movements were accurately portrayed. In comparison with the upper and lower body (fig.6.5), 85% say the upper body is better represented, while 10% say the lower body is better represented. Furthermore, 80% of the participants, with 5% reaching the scale extreme, stated that overall, they felt that their movements were unrealistic while in the virtual environment.

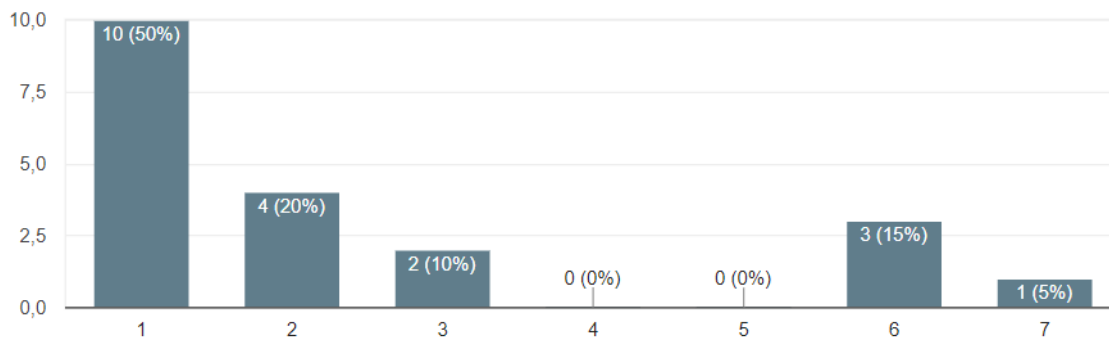


Figure 6.4: Less glitching comparison between Kinect (left side) and BlazePose (right side)

The fourth and final section of the questionnaire asked participants to indicate in a 7-point from 1 (strongly disagree), to 7 (strongly agree) their agreement or disagreement with the Kinect-BlazePose comparison(fig.6.6). Overall, 80% (50% chose the scale extreme) of the participants stated that they felt their body is more accurately represented by the avatar in Kinect (fig.6.2). With regard to latency (fig.6.1), 90% of participants rated the Kinect implementation as more responsive than BlazePose's. About 80% of respondents said that Kinect's implementation was less glitchy compared to BlazePose (with 50% choosing the maximum scale) (fig.6.4). Furthermore, 85% of respondents said that the Kinect implementation made their movement more realistic (fig.6.3).

6.2 Qualitative Data

A questionnaire with open-ended questions, some oral feedback, and observed actions during testing formed the qualitative data. The present section is divided into two categories so that data documentation can be more precise: oral and written feedback, and observed actions.

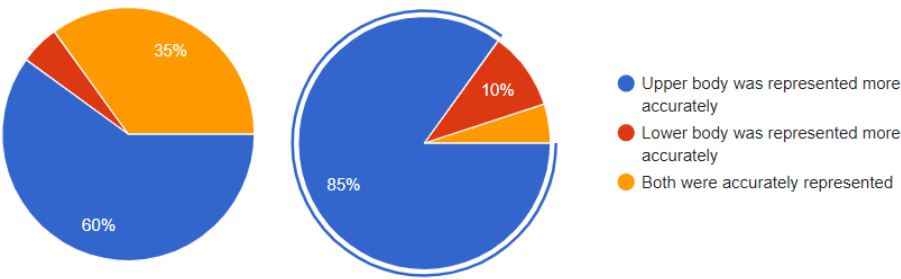


Figure 6.5: Upper-lower body accuracy comparison for Kinect (left side) and BlazePose (right side)

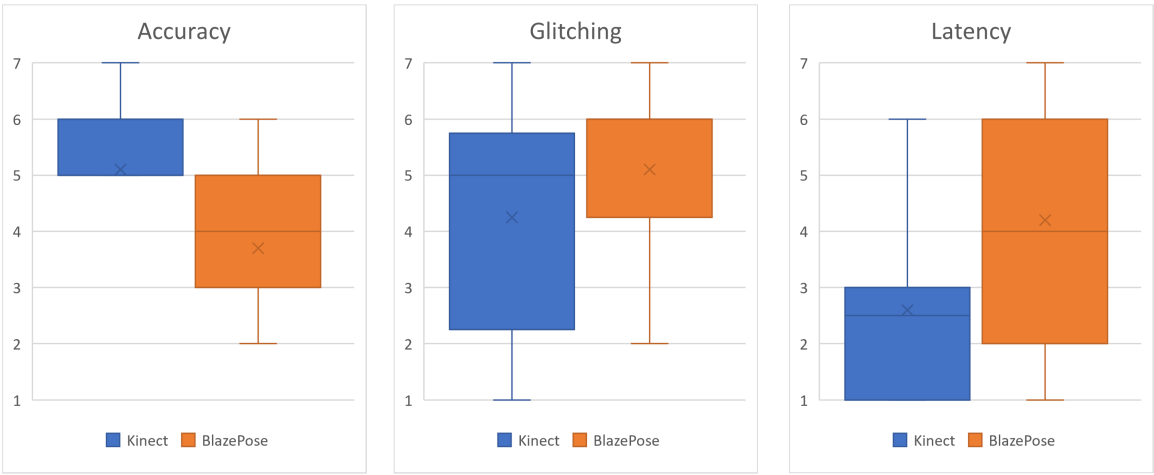


Figure 6.6: Impementations comparison: Accuracy, glitching, latency.

6.2.1 Oral and Written feedback

On the after testing questionnaire, open-stated questions were used to obtain written feedback on the tool. Some participants felt that the Kinect implementation was affected by latency issues in the lower body movement, while most did not notice any latency in their overall experience. A few participants who noticed some avatar glitching noted that the glitching was coming from the lower part of the body, while two of them noticed some glitching in their arms and hands. The majority of the participants who tested the BlazePose implementation felt the avatar's lower body had a latency during the test. Among them, one reported latency affecting moves in the avatar's whole body and two reported delay in avatar movements in its upper body. Almost all of the participants who noticed some glitching in the avatar's movement commented that it was mostly on the lower body, while three said there was also glitching on the upper body. The Kinect implementation throughout the questionnaire felt extremely realistic, though some participants noticed some glitches on the ground as they described in the general realistic movement open ended question. A participant described BlazePose implementation as being similar to the early Kinect implementation. There were some people who noticed their feet were hidden in the implementation of Kinect, with Kinect implementation still being more accurate than the BlazePose.

Finally, when asked about their overall experience, most of the participants stated that the tool is pretty fun and has lots of potential. Many of them liked the skeleton avatar while one stated that they would have preferred a different one. There were also a couple of comments stating that the experience was smooth and accurate, except for some glitching parts.

6.2.2 Observed Actions

Most of the participants have previously used Oculus Quest 2 or other head-mounted displays, so they were more comfortable with using and wearing them, while others were trying it for the very first time, so they needed a bit more time to engage with the VR experience. Experienced HMD users seemed to be thrilled when they realized that they could move their entire bodies through their skeleton avatars. Several of them commented on how great it was to see a virtual representation of themselves, making nice comments about it when they first interacted with them. We received a lot of feedback while testing the tool, which was reflected in the open ended questions of the questionnaire. Participants took some time after the testing procedure to interact freely with their avatars, performing different types of moving routines such as dancing and fighting, and they seemed cheerful when the session ended.

Chapter 7

Discussion

7.1 Methodology

Gathered qualitative data and answers to open-ended questions help in increasing the quality and value of the conducted study and user tests, since the low number of test subjects may have a negative effect on the reliability of results.

The conducted user tests were simplistic, and have been designed so that they could be completed in a short window of time by each subject. In order to achieve a broader and more precise results the testing procedure would require much more complex preparation. The virtual environments in which the test movement interaction were conducted could have been made to resemble a more realistic room for example one that would resemble a gym interior. Such approach could have influenced the degree of immersion of the subjects.

The tool created during the completion of this study was only assessed subjectively by the test group. The significance of the study could be increased by conducting a range of objective tests involving a comparison between the obtained human joints map and their actually physical counterparts. Inclusion of a professional grade tool like Rokoko suit in the comparison could also render the result more scientifically valuable.

7.2 User Feedback

Feedback provided by the users who have tested the created application has been valuable and provided interesting insights on its performance of the application. Most participants have reported that their body movements are being reproduced accurately. The result of the Kinect implementation was rated high in terms of accuracy and less latency was noticed compared with the BlazePose implementation. In terms of the movement direction subjects have assessed that horizontal and vertical movements were more precisely reproduced when compared to the

movements on the back-forth axis. One of the observed weak points of the Kinect based pose estimation was the issue with users' feet being clipped through the virtual floor. This issue was more common among shorter participants. In both implementations, most subjects found the upper body to be represented more accurately than the lower body.

Chapter 8

Conclusion

The goal of the project was to create an accessible solution for full body tracking, an objective that now has been fulfilled. Two different pose estimations tools were implemented and assessed by a group of test subjects. Results obtained from solution based on the depth camera (Kinect) has been shown to be preferred in term of the quality of its results over a computer vision based solution (BlazePose).

As a result of the carried out test a number of weaknesses and strengths have been observed. One of the limitations of the test solutions was occlusion. When using a single point of view to analyse a movement in three dimensional space an occlusion is almost impossible to avoid. Another issue was the slow response time of the neural network based tool.

The overall result of the motion tracking were mostly satisfactory. Participants were impressed that process as complex as full body pose estimation was achievable with no more than a camera and a personal computer.

8.1 Future work

The created tool could be possibly used as a foundation for more complex applications making use of the implemented full body tracking. Some of those examples could include: interactive applications, sport games or VR social platforms. Interactive applications like workplace training, interactive marketing campaigns, yoga or physio therapy. Sport training applications measuring performance or games e.g. football penalty shots or a VR obstacle course. Virtual social platforms such as Microsoft Mesh for Teams, Meta Horizons, Roblox or VR chat.

When considering social interactions in VR, full body tracking offers a range of advantages. One of them is the ability to utilize non verbal communication and body language. This possibility could have a positive impact on the user presence within the VR environment. In order to enhance the non verbal communication capabilities further the tool could benefit from implementing a solution for face

tracking.

There is space for improvement within the applications. Hopefully with more efficient machine learning algorithms, the latency could be minimised and the response time could be improved. With improved performance the solution could also be ported in to mobile platforms such as smartphones so that the full body tracking could be more portable and not bounded to stationary computers any more.

Bibliography

- [1] Mohd Hezri Amir et al. "Duke: enhancing virtual reality based FPS game with full-body interactions". In: *Proceedings of the 13th International Conference on Advances in Computer Entertainment Technology*. 2016, pp. 1–6.
- [2] Valentin Bazarevsky et al. "Blazepose: On-device real-time body pose tracking". In: *arXiv preprint arXiv:2006.10204* (2020).
- [3] Oculus Blog. "Introducing Oculus air link, A wireless way to play PC VR games on Oculus Quest 2, plus infinite office updates, support for 120 Hz on Quest 2, and more". In: <https://developer.oculus.com/> (Retrieved 2022-04-22).
- [4] Z. Cao et al. "OpenPose: Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2019).
- [5] Ross A Clark et al. "Three-dimensional cameras and skeleton pose tracking for physical function assessment: A review of uses, validity, current developments and Kinect alternatives". In: *Gait & posture* 68 (2019), pp. 193–200.
- [6] John Haas. "A history of the unity game engine". In: *Diss. Worcester Polytechnic Institute* (2014).
- [7] Jungong Han et al. "Enhanced Computer Vision With Microsoft Kinect Sensor: A Review". In: *IEEE Transactions on Cybernetics* 43.5 (2013), pp. 1318–1334. DOI: 10.1109/TCYB.2013.2265378.
- [8] Akiya Research Institute. URL: <https://vrlab.akiya-souken.co.jp/en/product> (visited on 05/22/2022).
- [9] Branislav Kisacanin, Vladimir Pavlovic, and Thomas S Huang. *Real-time vision for human-computer interaction*. Springer Science & Business Media, 2005.
- [10] Amit Krishan Kumar, Abhishek Kaushal Kumar, and Shuli Guo. "Two view-points based real-time recognition for hand gestures". In: *IET Image Processing* 14.17 (2020), pp. 4606–4613.
- [11] Jaron Lanier. "Virtually there". In: *Scientific American* 284.4 (2001), pp. 66–75.

- [12] D Lee et al. "A development of virtual reality game utilizing kinect, oculus rift and smartphone". In: *International Journal of Applied Engineering Research* 11.2 (2016), pp. 829–833.
- [13] Unity Manual. URL: <https://docs.unity3d.com/Manual/InverseKinematics.html> (visited on 05/22/2022).
- [14] Microsoft. URL: <https://docs.microsoft.com/en-us/windows/apps/design/devices/kinect-for-windows> (visited on 05/23/2022).
- [15] Microsoft. URL: <https://www.microsoft.com/en-us/download/details.aspx?id=44561>.
- [16] Senay Mihcin et al. "Investigation of wearable motion capture system towards biomechanical modelling". In: *2019 IEEE International Symposium on Medical Measurements and Applications (MeMeA)*. IEEE. 2019, pp. 1–5.
- [17] Matjaž Mihelj, Domen Novak, and Samo Beguš. "Virtual reality technology and applications". In: (2014).
- [18] Shaleph O'Neill. *Interactive media: The semiotics of embodied interaction*. Springer Science & Business Media, 2008.
- [19] Mehdi Ousmer, Jean Vanderdonckt, and Sabin Buraga. "An ontology for reasoning on body-based gestures". In: *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. 2019, pp. 1–6.
- [20] Dhaval Parmar. "Evaluating the effects of immersive embodied interaction on cognition in virtual reality". PhD thesis. Clemson University, 2017.
- [21] Github repository. URL: <https://github.com/digital-standard/ThreeDPoseUnityBarracuda> (visited on 05/22/2022).
- [22] Github repository. URL: <https://github.com/creativeIKEP/BlazePoseBarracuda> (visited on 05/23/2022).
- [23] Google Research. URL: <https://ai.googleblog.com/2020/08/on-device-real-time-body-pose-tracking.html> (visited on 05/23/2022).
- [24] Ulrike Schultze. "Embodiment and presence in virtual worlds: a review". In: *Journal of Information Technology* 25.4 (2010), pp. 434–449.
- [25] Misha Sra and Chris Schmandt. "Metaspace: Full-body tracking for immersive multiperson virtual reality". In: *Adjunct Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. 2015, pp. 47–48.
- [26] Thomas Stranick and Christian Lopez. "Adaptive Virtual Reality Exergame: Promoting Physical Activity Among Workers". In: *Journal of Computing and Information Science in Engineering* 22.3 (2022).

- [27] Polytope Studio. URL: <https://assetstore.unity.com/packages/3d/characters/humanoids/fantasy/lowpoly-medieval-skeleton-free-pack-18188> (visited on 05/23/2022).
- [28] Meta Technologies. URL: <https://about.facebook.com/> (visited on 05/23/2022).
- [29] Unity Technologies. URL: <https://unity.com/> (visited on 05/23/2022).
- [30] TensorFlow. URL: <https://blog.tensorflow.org/2018/05/real-time-human-pose-estimation-in.html> (visited on 05/22/2022).
- [31] TensorFlow. URL: <https://blog.tensorflow.org/2021/05/next-generation-pose-detection-with-movenet-and-tensorflowjs.html>.
- [32] Wikipedia. URL: <https://en.wikipedia.org/wiki/Kinect> (visited on 05/22/2022).
- [33] Di Wu and Da-Wen Sun. "Colour measurements by computer vision for food quality control—A review". In: *Trends in Food Science & Technology* 29.1 (2013), pp. 5–20.
- [34] Jingming Xie. "Research on key technologies base Unity3D game engine". In: *2012 7th international conference on computer science & education (ICCSE)*. IEEE, 2012, pp. 695–699.
- [35] Jackie Yang et al. "HybridTrak: Adding Full-Body Tracking to VR Using an Off-the-Shelf Webcam". In: *CHI Conference on Human Factors in Computing Systems*. 2022, pp. 1–13.
- [36] Yuki2022. URL: <https://assetstore.unity.com/packages/2d/textures-materials/sky/free-stylized-skybox-212257> (visited on 05/23/2022).
- [37] Zhengyou Zhang. "Microsoft kinect sensor and its effect". In: *IEEE multimedia* 19.2 (2012), pp. 4–10.
- [38] Ning-Ning Zhou and Yu-Long Deng. "Virtual reality: A state-of-the-art survey". In: *International Journal of Automation and Computing* 6.4 (2009), pp. 319–325.
- [39] Yimin Zhou, Guolai Jiang, and Yaorong Lin. "A novel finger and hand pose estimation technique for real-time hand gesture recognition". In: *Pattern Recognition* 49 (2016), pp. 102–114.

Appendix A

Scripts

A.1 VNectModel.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

/// <summary>
/// Position index of joint point
/// </summary>
public enum PositionIndex : int
{
    Nose = 0,
    lEyeInner,
    lEye,
    lEyeOuter,
    rEyeInner,
    rEye,
    rEyeOuter,
    lEar,
    rEar,
    mouthL,
    mouthR,
    lShoulder,
    rShoulder,
    lElbow,
    rElbow,
    lWrist,
    rWrist,
```

```

    lPinky,
    rPinky,
    lIndex,
    rIndex,
    lThumb,
    rThumb,
    lHip,
    rHip,
    lKnee,
    rKnee,
    lAnkle,
    rAnkle,
    lHeel,
    rHeel,
    lFootIndex,
    rFootIndex,
    humanVisible,

    //Calculated coordinates
    head,
    neck,
    chest,
    spine,
    hips,
    lController,
    rController,
    lPhantomElbow,
    rPhantomElbow,
    centerHead,
    phantomNose,

    Count,
    None,
}

public static partial class EnumExtend
{
    public static int Int(this PositionIndex i)
    {
        return (int)i;
    }
}

public class VNectModel : MonoBehaviour
{
    public class JointPoint

```



```

{
    public Vector3 Pos3D = new Vector3();
    public float score3D;

    // Bones
    public Transform Transform = null;
    public Quaternion InitRotation;
    public Quaternion Inverse;
    public Quaternion InverseRotation;

    public JointPoint Child = null;
    public JointPoint Parent = null;
}

public class Skeleton
{
    public GameObject LineObject;
    public LineRenderer Line;
    public JointPoint start = null;
    public JointPoint end = null;
}

private List<Skeleton> Skeletons = new List<Skeleton>();
public Material SkeletonMaterial;
public bool ShowSkeleton;
private bool useSkeleton;
public float SkeletonX;
public float SkeletonY;
public float SkeletonZ;
public float SkeletonScale;

// Joint position and bone
private JointPoint[] jointPoints;
public JointPoint[] JointPoints { get { return jointPoints; } }
private Vector3 initPosition; // Initial center position
private Vector3 jointPositionOffset = Vector3.zero;

// Avatar
public GameObject ModelObject;
public GameObject Nose;
private Animator anim;
private Vector3 avatarDimensions;
private Vector3 avatarCenter;

// HMD
private InputDevice hmdDevice;
private InputDevice leftController;

```

```

private InputDevice rightController;
private bool lastPrimaryButtonValue = false;
private bool vrRunning = false;
private Vector3 hmdPosition;
private Vector3 leftControllerPosition;
private Vector3 rightControllerPosition;
private Quaternion hmdRotation;
private Quaternion leftControllerRotation;
private Quaternion rightControllerRotation;

private string sceneName;
private bool kinectScene = false;
public PoseVisualizer3D PoseVisualizer3D;
public FaceManager FaceManager;
public BodySourceView BodySourceView;
public GameObject Instruction;
private bool displayText = false;

void Awake()
{
    sceneName = SceneManager.GetActiveScene().name;

    if (sceneName == "KinectScene")
        kinectScene = true;

    Instruction.SetActive(displayText);
}

private void Update()
{
    if (kinectScene)
    {
        if (!vrRunning)
        {
            // Head rotation
            jointPoints[PositionIndex.head.Int()].Transform.rotation =
                FaceManager.GetFaceRotation();
        }
        else
        {
            // Phantom nose position
            jointPoints[PositionIndex.phantomNose.Int()].Pos3D = new
                Vector3(jointPoints[PositionIndex.centerHead.Int()].Pos3D.x,
                    jointPoints[PositionIndex.centerHead.Int()].Pos3D.y,
                    jointPoints[PositionIndex.centerHead.Int()].Pos3D.z +
                    0.1f);
        }
    }
}

```

```
    }
}

if (hmdDevice.isValid)
{
    hmdDevice.TryGetFeatureValue(CommonUsages.devicePosition, out
        hmdPosition);
    hmdDevice.TryGetFeatureValue(CommonUsages.deviceRotation, out
        hmdRotation);
}
if (leftController.isValid)
{
    leftController.TryGetFeatureValue(CommonUsages.devicePosition,
        out leftControllerPosition);
    leftController.TryGetFeatureValue(CommonUsages.deviceRotation,
        out leftControllerRotation);
}
if (rightController.isValid)
{
    rightController.TryGetFeatureValue(CommonUsages.devicePosition, out
        rightControllerPosition);
    rightController.TryGetFeatureValue(CommonUsages.deviceRotation,
        out rightControllerRotation);
}

if(vrRunning)
{
    // Head rotation
    jointPoints[PositionIndex.head.Int()].Transform.rotation =
        hmdRotation;
    // Wrists positions
    if (!kinectScene)
    {
        jointPoints[PositionIndex.lController.Int()].Pos3D =
            leftControllerPosition - hmdPosition +
            jointPoints[PositionIndex.Nose.Int()].Pos3D;
        jointPoints[PositionIndex.rController.Int()].Pos3D =
            rightControllerPosition - hmdPosition +
            jointPoints[PositionIndex.Nose.Int()].Pos3D;
    }
    else
    {
        jointPoints[PositionIndex.lController.Int()].Pos3D =
            leftControllerPosition - hmdPosition +
            jointPoints[PositionIndex.phantomNose.Int()].Pos3D;
        jointPoints[PositionIndex.rController.Int()].Pos3D =
            rightControllerPosition - hmdPosition +
```

```

        jointPoints[PositionIndex.phantomNose.Int()].Pos3D;
    }
    // Wrists rotations
    jointPoints[PositionIndex.lController.Int()].Transform.rotation
        = leftControllerRotation * Quaternion.Euler(new
            Vector3(-180f, -90f, -80f));
    jointPoints[PositionIndex.rController.Int()].Transform.rotation
        = rightControllerRotation * Quaternion.Euler(new Vector3(0f,
            90f, -80f));
    // Left elbow position
    Vector3 lElbowProjection =
        Vector3.Project((jointPoints[PositionIndex.lElbow.Int()].Pos3D
            - jointPoints[PositionIndex.lShoulder.Int()].Pos3D),
            (jointPoints[PositionIndex.lController.Int()].Pos3D -
                jointPoints[PositionIndex.lShoulder.Int()].Pos3D)) +
            jointPoints[PositionIndex.lShoulder.Int()].Pos3D;
    Vector3 lElbowProjectionElbow =
        jointPoints[PositionIndex.lElbow.Int()].Pos3D -
            lElbowProjection;
    Vector3 lPhantomElbowProjection =
        Vector3.Lerp(jointPoints[PositionIndex.lShoulder.Int()].Pos3D,
            jointPoints[PositionIndex.lController.Int()].Pos3D, 0.5f);
    jointPoints[PositionIndex.lPhantomElbow.Int()].Pos3D =
        lPhantomElbowProjection + lElbowProjectionElbow;
    // Right elbow position
    Vector3 rElbowProjection =
        Vector3.Project((jointPoints[PositionIndex.rElbow.Int()].Pos3D
            - jointPoints[PositionIndex.rShoulder.Int()].Pos3D),
            (jointPoints[PositionIndex.rController.Int()].Pos3D -
                jointPoints[PositionIndex.rShoulder.Int()].Pos3D)) +
            jointPoints[PositionIndex.rShoulder.Int()].Pos3D;
    Vector3 rElbowProjectionElbow =
        jointPoints[PositionIndex.rElbow.Int()].Pos3D -
            rElbowProjection;
    Vector3 rPhantomElbowProjection =
        Vector3.Lerp(jointPoints[PositionIndex.rShoulder.Int()].Pos3D,
            jointPoints[PositionIndex.rController.Int()].Pos3D, 0.5f);
    jointPoints[PositionIndex.rPhantomElbow.Int()].Pos3D =
        rPhantomElbowProjection + rElbowProjectionElbow;
}

// Primary button on left controller triggers calibration
bool primaryButtonValue = false;
if
    (UnityEngine.InputSystem.Keyboard.current.spaceKey.wasPressedThisFrame
    ||
    (leftController.TryGetFeatureValue(CommonUsages.primaryButton,

```

```

        out primaryButtonValue) && primaryButtonValue !=
        lastPrimaryButtonValue && primaryButtonValue))
        RunCalibration();
    lastPrimaryButtonValue = primaryButtonValue;

    if (jointPoints != null)
        PoseUpdate();
}

/// <summary>
/// Initialize joint points
/// </summary>
/// <returns></returns>
public JointPoint[] Initialize()
{
    vrRunning = isVrRunning();
    hmdDevice = InputDevices.GetDeviceAtXRNode(XRNode.CenterEye);
    leftController = InputDevices.GetDeviceAtXRNode(XRNode.LeftHand);
    rightController = InputDevices.GetDeviceAtXRNode(XRNode.RightHand);

    jointPoints = new JointPoint[PositionIndex.Count.Int()];
    for (var i = 0; i < PositionIndex.Count.Int(); i++)
        jointPoints[i] = new JointPoint();

    anim = ModelObject.GetComponent<Animator>();

    avatarDimensions.x =
        Vector3.Distance(anim.GetBoneTransform(HumanBodyBones.RightHand).position,
            anim.GetBoneTransform(HumanBodyBones.LeftHand).position);
    avatarDimensions.y = Nose.transform.position.y;
    avatarCenter = GetCenter(gameObject);

    // Right Arm
    jointPoints[PositionIndex.rShoulder.Int()].Transform =
        anim.GetBoneTransform(HumanBodyBones.RightUpperArm);
    if (!vrRunning)
    {
        jointPoints[PositionIndex.rElbow.Int()].Transform =
            anim.GetBoneTransform(HumanBodyBones.RightLowerArm);
        jointPoints[PositionIndex.rWrist.Int()].Transform =
            anim.GetBoneTransform(HumanBodyBones.RightHand);
        jointPoints[PositionIndex.rThumb.Int()].Transform =
            anim.GetBoneTransform(HumanBodyBones.RightThumbIntermediate);
        jointPoints[PositionIndex.rPinky.Int()].Transform =
            anim.GetBoneTransform(HumanBodyBones.RightLittleIntermediate);
    }
    else

```

```

{
    jointPoints[PositionIndex.rPhantomElbow.Int()].Transform =
        anim.GetBoneTransform(HumanBodyBones.RightLowerArm);
    jointPoints[PositionIndex.rController.Int()].Transform =
        anim.GetBoneTransform(HumanBodyBones.RightHand);
}

// Left Arm
jointPoints[PositionIndex.lShoulder.Int()].Transform =
    anim.GetBoneTransform(HumanBodyBones.LeftUpperArm);
if(!vrRunning)
{
    jointPoints[PositionIndex.lElbow.Int()].Transform =
        anim.GetBoneTransform(HumanBodyBones.LeftLowerArm);
    jointPoints[PositionIndex.lWrist.Int()].Transform =
        anim.GetBoneTransform(HumanBodyBones.LeftHand);
    jointPoints[PositionIndex.lThumb.Int()].Transform =
        anim.GetBoneTransform(HumanBodyBones.LeftThumbIntermediate);
    jointPoints[PositionIndex.lPinky.Int()].Transform =
        anim.GetBoneTransform(HumanBodyBones.LeftLittleIntermediate);
}
else
{
    jointPoints[PositionIndex.lPhantomElbow.Int()].Transform =
        anim.GetBoneTransform(HumanBodyBones.LeftLowerArm);
    jointPoints[PositionIndex.lController.Int()].Transform =
        anim.GetBoneTransform(HumanBodyBones.LeftHand);
}

// Head
if (kinectScene && vrRunning)
    jointPoints[PositionIndex.phantomNose.Int()].Transform =
        Nose.transform;
else
{
    jointPoints[PositionIndex.lEar.Int()].Transform =
        anim.GetBoneTransform(HumanBodyBones.Head);
    jointPoints[PositionIndex.lEye.Int()].Transform =
        anim.GetBoneTransform(HumanBodyBones.LeftEye);
    jointPoints[PositionIndex.rEar.Int()].Transform =
        anim.GetBoneTransform(HumanBodyBones.Head);
    jointPoints[PositionIndex.rEye.Int()].Transform =
        anim.GetBoneTransform(HumanBodyBones.RightEye);
    jointPoints[PositionIndex.Nose.Int()].Transform = Nose.transform;
}

// Right Leg

```

```

jointPoints[PositionIndex.rHip.Int()].Transform =
    anim.GetBoneTransform(HumanBodyBones.RightUpperLeg);
jointPoints[PositionIndex.rKnee.Int()].Transform =
    anim.GetBoneTransform(HumanBodyBones.RightLowerLeg);
jointPoints[PositionIndex.rAnkle.Int()].Transform =
    anim.GetBoneTransform(HumanBodyBones.RightFoot);
jointPoints[PositionIndex.rFootIndex.Int()].Transform =
    anim.GetBoneTransform(HumanBodyBones.RightToes);
// Left Leg
jointPoints[PositionIndex.lHip.Int()].Transform =
    anim.GetBoneTransform(HumanBodyBones.LeftUpperLeg);
jointPoints[PositionIndex.lKnee.Int()].Transform =
    anim.GetBoneTransform(HumanBodyBones.LeftLowerLeg);
jointPoints[PositionIndex.lAnkle.Int()].Transform =
    anim.GetBoneTransform(HumanBodyBones.LeftFoot);
jointPoints[PositionIndex.lFootIndex.Int()].Transform =
    anim.GetBoneTransform(HumanBodyBones.LeftToes);

// Spine
jointPoints[PositionIndex.head.Int()].Transform =
    anim.GetBoneTransform(HumanBodyBones.Head);
jointPoints[PositionIndex.neck.Int()].Transform =
    anim.GetBoneTransform(HumanBodyBones.Neck);
jointPoints[PositionIndex.chest.Int()].Transform =
    anim.GetBoneTransform(HumanBodyBones.Chest);
jointPoints[PositionIndex.spine.Int()].Transform =
    anim.GetBoneTransform(HumanBodyBones.Spine);
jointPoints[PositionIndex.hips.Int()].Transform =
    anim.GetBoneTransform(HumanBodyBones.Hips);

// Parent-Child Setup
// Right Arm
if(!vrRunning)
{
    jointPoints[PositionIndex.rShoulder.Int()].Child =
        jointPoints[PositionIndex.rElbow.Int()];
    jointPoints[PositionIndex.rElbow.Int()].Child =
        jointPoints[PositionIndex.rWrist.Int()];
    jointPoints[PositionIndex.rElbow.Int()].Parent =
        jointPoints[PositionIndex.rShoulder.Int()];
}
else
{
    jointPoints[PositionIndex.rShoulder.Int()].Child =
        jointPoints[PositionIndex.rPhantomElbow.Int()];
    jointPoints[PositionIndex.rPhantomElbow.Int()].Child =
        jointPoints[PositionIndex.rController.Int()];
}

```

```

        jointPoints[PositionIndex.rPhantomElbow.Int()].Parent =
            jointPoints[PositionIndex.rShoulder.Int()];
    }

    // Left Arm
    if(!vrRunning)
    {
        jointPoints[PositionIndex.lShoulder.Int()].Child =
            jointPoints[PositionIndex.lElbow.Int()];
        jointPoints[PositionIndex.lElbow.Int()].Child =
            jointPoints[PositionIndex.lWrist.Int()];
        jointPoints[PositionIndex.lElbow.Int()].Parent =
            jointPoints[PositionIndex.lShoulder.Int()];
    }
    else
    {
        jointPoints[PositionIndex.lShoulder.Int()].Child =
            jointPoints[PositionIndex.lPhantomElbow.Int()];
        jointPoints[PositionIndex.lPhantomElbow.Int()].Child =
            jointPoints[PositionIndex.lController.Int()];
        jointPoints[PositionIndex.lPhantomElbow.Int()].Parent =
            jointPoints[PositionIndex.lShoulder.Int()];
    }

    // Right Leg
    jointPoints[PositionIndex.rHip.Int()].Child =
        jointPoints[PositionIndex.rKnee.Int()];
    jointPoints[PositionIndex.rKnee.Int()].Child =
        jointPoints[PositionIndex.rAnkle.Int()];
    jointPoints[PositionIndex.rAnkle.Int()].Child =
        jointPoints[PositionIndex.rFootIndex.Int()];
    jointPoints[PositionIndex.rAnkle.Int()].Parent =
        jointPoints[PositionIndex.rKnee.Int()];
    // Left Leg
    jointPoints[PositionIndex.lHip.Int()].Child =
        jointPoints[PositionIndex.lKnee.Int()];
    jointPoints[PositionIndex.lKnee.Int()].Child =
        jointPoints[PositionIndex.lAnkle.Int()];
    jointPoints[PositionIndex.lAnkle.Int()].Child =
        jointPoints[PositionIndex.lFootIndex.Int()];
    jointPoints[PositionIndex.lAnkle.Int()].Parent =
        jointPoints[PositionIndex.lKnee.Int()];

    // Spine
    jointPoints[PositionIndex.spine.Int()].Child =
        jointPoints[PositionIndex.chest.Int()];

```



```
jointPoints[PositionIndex.chest.Int()].Child =
    jointPoints[PositionIndex.neck.Int()];
jointPoints[PositionIndex.neck.Int()].Child =
    jointPoints[PositionIndex.head.Int()];

useSkeleton = ShowSkeleton;
if (useSkeleton)
{
    // Skeleton Lines
    // Right Arm
    if(!vrRunning)
    {
        AddSkeleton(PositionIndex.rShoulder, PositionIndex.rElbow);
        AddSkeleton(PositionIndex.rElbow, PositionIndex.rWrist);
        AddSkeleton(PositionIndex.rWrist, PositionIndex.rThumb);
        AddSkeleton(PositionIndex.rWrist, PositionIndex.rPinky);
    }
    else
    {
        AddSkeleton(PositionIndex.rShoulder,
            PositionIndex.rPhantomElbow);
        AddSkeleton(PositionIndex.rPhantomElbow,
            PositionIndex.rController);
    }

    // Left Arm
    if(!vrRunning)
    {
        AddSkeleton(PositionIndex.lShoulder, PositionIndex.lElbow);
        AddSkeleton(PositionIndex.lElbow, PositionIndex.lWrist);
        AddSkeleton(PositionIndex.lWrist, PositionIndex.lThumb);
        AddSkeleton(PositionIndex.lWrist, PositionIndex.lPinky);
    }
    else
    {
        AddSkeleton(PositionIndex.lShoulder,
            PositionIndex.lPhantomElbow);
        AddSkeleton(PositionIndex.lPhantomElbow,
            PositionIndex.lController);
    }

    // Head
    if (kinectScene && vrRunning)
        AddSkeleton(PositionIndex.centerHead,
            PositionIndex.phantomNose);
    else
    {
```

```

        AddSkeleton(PositionIndex.lEar, PositionIndex.lEye);
        AddSkeleton(PositionIndex.lEye, PositionIndex.Nose);
        AddSkeleton(PositionIndex.rEar, PositionIndex.rEye);
        AddSkeleton(PositionIndex.rEye, PositionIndex.Nose);
    }

    // Right Leg
    AddSkeleton(PositionIndex.rHip, PositionIndex.rKnee);
    AddSkeleton(PositionIndex.rKnee, PositionIndex.rAnkle);
    AddSkeleton(PositionIndex.rAnkle, PositionIndex.rFootIndex);
    // Left Leg
    AddSkeleton(PositionIndex.lHip, PositionIndex.lKnee);
    AddSkeleton(PositionIndex.lKnee, PositionIndex.lAnkle);
    AddSkeleton(PositionIndex.lAnkle, PositionIndex.lFootIndex);

    // Torso
    AddSkeleton(PositionIndex.hips, PositionIndex.spine);
    AddSkeleton(PositionIndex.spine, PositionIndex.chest);
    AddSkeleton(PositionIndex.chest, PositionIndex.neck);
    AddSkeleton(PositionIndex.neck, PositionIndex.head);
    AddSkeleton(PositionIndex.chest, PositionIndex.rShoulder);
    AddSkeleton(PositionIndex.chest, PositionIndex.lShoulder);
    AddSkeleton(PositionIndex.hips, PositionIndex.rHip);
    AddSkeleton(PositionIndex.hips, PositionIndex.lHip);
}

// Set Inverse
var forward =
    TriangleNormal(jointPoints[PositionIndex.hips.Int()].Transform.position,
        jointPoints[PositionIndex.lHip.Int()].Transform.position,
        jointPoints[PositionIndex.rHip.Int()].Transform.position);
foreach (var jointPoint in jointPoints)
{
    if (jointPoint != null)
    {
        if (jointPoint.Transform != null)
        {
            jointPoint.InitRotation = jointPoint.Transform.rotation;
        }
        if (jointPoint.Child != null && jointPoint.Child.Transform !=
            null && jointPoint.Child.Transform.position != null)
        {
            jointPoint.Inverse = GetInverse(jointPoint,
                jointPoint.Child, forward);
            jointPoint.InverseRotation = jointPoint.Inverse *
                jointPoint.InitRotation;
        }
    }
}

```

```

    }
}

// Hips Rotation
var hips = jointPoints[PositionIndex.hips.Int()];
initPosition =
    jointPoints[PositionIndex.hips.Int()].Transform.position;
hips.Inverse = Quaternion.Inverse(Quaternion.LookRotation(forward));
hips.InverseRotation = hips.Inverse * hips.InitRotation;

// Head Rotation
var head = jointPoints[PositionIndex.head.Int()];
head.InitRotation =
    jointPoints[PositionIndex.head.Int()].Transform.rotation;
if(kinectScene && vrRunning)
{
    var gaze =
        jointPoints[PositionIndex.phantomNose.Int()].Transform.position
        - jointPoints[PositionIndex.head.Int()].Transform.position;
    head.Inverse = Quaternion.Inverse(Quaternion.LookRotation(gaze));
    head.InverseRotation = head.Inverse * head.InitRotation;
}
else
{
    var gaze =
        jointPoints[PositionIndex.Nose.Int()].Transform.position -
        jointPoints[PositionIndex.head.Int()].Transform.position;
    head.Inverse = Quaternion.Inverse(Quaternion.LookRotation(gaze));
    head.InverseRotation = head.Inverse * head.InitRotation;
}

if(!vrRunning)
{
    // Wrists rotation
    var lWrist = jointPoints[PositionIndex.lWrist.Int()];
    var lf = TriangleNormal(lWrist.Pos3D,
        jointPoints[PositionIndex.lPinky.Int()].Pos3D,
        jointPoints[PositionIndex.lThumb.Int()].Pos3D);
    lWrist.InitRotation = lWrist.Transform.rotation;
    lWrist.Inverse =
        Quaternion.Inverse(Quaternion.LookRotation(jointPoints[PositionIndex.lThumb.Int()].Transform.position -
        jointPoints[PositionIndex.lPinky.Int()].Transform.position,
        lf));
    lWrist.InverseRotation = lWrist.Inverse * lWrist.InitRotation;

    var rWrist = jointPoints[PositionIndex.rWrist.Int()];

```

```

        var rf = TriangleNormal(rWrist.Pos3D,
                                jointPoints[PositionIndex.rThumb.Int()].Pos3D,
                                jointPoints[PositionIndex.rPinky.Int()].Pos3D);
        rWrist.InitRotation =
            jointPoints[PositionIndex.rWrist.Int()].Transform.rotation;
        rWrist.Inverse =
            Quaternion.Inverse(Quaternion.LookRotation(jointPoints[PositionIndex.rThumb.Int()].Pos3D -
                jointPoints[PositionIndex.rPinky.Int()].Transform.position,
                rf));
        rWrist.InverseRotation = rWrist.Inverse * rWrist.InitRotation;
    }

    return JointPoints;
}

public void PoseUpdate()
{
    // movement and rotation of the center
    var forward =
        TriangleNormal(jointPoints[PositionIndex.hips.Int()].Pos3D,
                        jointPoints[PositionIndex.lHip.Int()].Pos3D,
                        jointPoints[PositionIndex.rHip.Int()].Pos3D);
    if(!vrRunning)
        jointPoints[PositionIndex.hips.Int()].Transform.position =
            jointPoints[PositionIndex.hips.Int()].Pos3D + initPosition -
            jointPositionOffset;
    else
    {
        if (kinectScene)
            jointPoints[PositionIndex.hips.Int()].Transform.position =
                jointPoints[PositionIndex.hips.Int()].Pos3D -
                jointPoints[PositionIndex.phantomNose.Int()].Pos3D +
                hmdPosition - jointPositionOffset;
        else
            jointPoints[PositionIndex.hips.Int()].Transform.position =
                jointPoints[PositionIndex.hips.Int()].Pos3D -
                jointPoints[PositionIndex.Nose.Int()].Pos3D + hmdPosition
                - jointPositionOffset;
    }
    jointPoints[PositionIndex.hips.Int()].Transform.rotation =
        Quaternion.LookRotation(forward) *
        jointPoints[PositionIndex.hips.Int()].InverseRotation;

    // rotation of each of the bones
    foreach (var jointPoint in jointPoints)
    {

```

```

    if (jointPoint.Parent != null)
    {
        var fv = jointPoint.Parent.Pos3D - jointPoint.Pos3D;
        jointPoint.Transform.rotation =
            Quaternion.LookRotation(jointPoint.Pos3D -
                jointPoint.Child.Pos3D, fv) * jointPoint.InverseRotation;
    }
    else if (jointPoint.Child != null)
    {
        jointPoint.Transform.rotation =
            Quaternion.LookRotation(jointPoint.Pos3D -
                jointPoint.Child.Pos3D, forward) *
                jointPoint.InverseRotation;
    }
}

if(!vrRunning || (!kinectScene && !vrRunning))
{
    // Head Rotation
    var gaze = jointPoints[PositionIndex.Nose.Int()].Pos3D -
        jointPoints[PositionIndex.head.Int()].Pos3D;
    var f =
        TriangleNormal(jointPoints[PositionIndex.Nose.Int()].Pos3D,
            jointPoints[PositionIndex.rEar.Int()].Pos3D,
            jointPoints[PositionIndex.lEar.Int()].Pos3D);
    var head = jointPoints[PositionIndex.head.Int()];
    head.Transform.rotation = Quaternion.LookRotation(gaze, f) *
        head.InverseRotation;
}

if(!vrRunning)
{
    // Wrist rotation
    var lWrist = jointPoints[PositionIndex.lWrist.Int()];
    var lf = TriangleNormal(lWrist.Pos3D,
        jointPoints[PositionIndex.lPinky.Int()].Pos3D,
        jointPoints[PositionIndex.lThumb.Int()].Pos3D);
    lWrist.Transform.rotation =
        Quaternion.LookRotation(jointPoints[PositionIndex.lThumb.Int()].Pos3D
            - jointPoints[PositionIndex.lPinky.Int()].Pos3D, lf) *
            lWrist.InverseRotation;

    var rWrist = jointPoints[PositionIndex.rWrist.Int()];
    var rf = TriangleNormal(rWrist.Pos3D,
        jointPoints[PositionIndex.rThumb.Int()].Pos3D,
        jointPoints[PositionIndex.rPinky.Int()].Pos3D);

```

```

        rWrist.Transform.rotation =
            Quaternion.LookRotation(jointPoints[PositionIndex.rThumb.Int()].Pos3D
            - jointPoints[PositionIndex.rPinky.Int()].Pos3D, rf) *
            rWrist.InverseRotation;
    }

    foreach (var sk in Skeletons)
    {
        var s = sk.start;
        var e = sk.end;

        sk.Line.SetPosition(0, new Vector3(s.Pos3D.x * SkeletonScale +
            SkeletonX, s.Pos3D.y * SkeletonScale + SkeletonY, s.Pos3D.z
            * SkeletonScale + SkeletonZ));
        sk.Line.SetPosition(1, new Vector3(e.Pos3D.x * SkeletonScale +
            SkeletonX, e.Pos3D.y * SkeletonScale + SkeletonY, e.Pos3D.z
            * SkeletonScale + SkeletonZ));
    }
}

Vector3 TriangleNormal(Vector3 a, Vector3 b, Vector3 c)
{
    Vector3 d1 = a - b;
    Vector3 d2 = a - c;

    Vector3 dd = Vector3.Cross(d1, d2);
    dd.Normalize();

    return dd;
}

private Quaternion GetInverse(JointPoint p1, JointPoint p2, Vector3
    forward)
{
    return
        Quaternion.Inverse(Quaternion.LookRotation(p1.Transform.position
            - p2.Transform.position, forward));
}

/// <summary>
/// Add skelton from joint points
/// </summary>
/// <param name="s">position index</param>
/// <param name="e">position index</param>
private void AddSkeleton(PositionIndex s, PositionIndex e)
{
    var sk = new Skeleton()

```

```

    {
        LineObject = new GameObject("Line"),
        start = jointPoints[s.Int()],
        end = jointPoints[e.Int()],
    };

    sk.Line = sk.LineObject.AddComponent<LineRenderer>();
    sk.Line.startWidth = 0.025f;
    sk.Line.endWidth = 0.005f;

    // define the number of vertex
    sk.Line.positionCount = 2;
    sk.Line.material = SkeletonMaterial;

    Skeletons.Add(sk);
}

private static bool isVrRunning()
{
    var xrDisplaySubsystems = new List<XRDisplaySubsystem>();
    SubsystemManager.GetInstances<XRDisplaySubsystem>(xrDisplaySubsystems);
    foreach (var xrDisplay in xrDisplaySubsystems)
    {
        if (xrDisplay.running)
        {
            return true;
        }
    }
    return false;
}

private Vector3 GetCenter(GameObject obj)
{
    Vector3 sumVector = Vector3.zero;

    foreach (Transform child in obj.transform)
    {
        sumVector += child.position;
    }

    Vector3 groupCenter = sumVector / obj.transform.childCount;
    return sumVector;
}

private void RunCalibration()
{
    Instruction.SetActive(true);
}

```

```

if (!vrRunning)
{
    Debug.Log("Avatar calibration will begin in 5 seconds, please
              stand in T-pose!");
    if (kinectScene)
    {
        StartCoroutine(BodySourceView.KinectCalibrationRoutine(vrRunning,
            kinectTDimensionsCalculated => {
                ScaleAvatar(kinectTDimensionsCalculated);
                Debug.Log("Avatar calibration done!");
                Instruction.SetActive(false);
            }));
    }
    else
    {
        StartCoroutine(PoseVisualizer3D.PoseCalibrationRoutine(vrRunning,
            poseTDimensionsCalculated => {
                ScaleAvatar(poseTDimensionsCalculated);
                Debug.Log("Avatar calibration done!");
                Instruction.SetActive(false);
            }));
    }
}
else
{
    Debug.Log("VR calibration will begin in 5 seconds, please stand
              in T-pose!");
    StartCoroutine(VrCalibrationRoutine(vrTDimensionsCalculated => {
        Vector3 vrTDimensions = vrTDimensionsCalculated;
        if (kinectScene)
        {
            StartCoroutine(BodySourceView.KinectCalibrationRoutine(vrRunning,
                kinectTDimensionsCalculated => {
                    BodySourceView.ScaleKinect(vrTDimensions,
                        kinectTDimensionsCalculated);
                    Debug.Log("VR calibration done!");
                    Instruction.SetActive(false);
                }));
        }
        else
        {
            StartCoroutine(PoseVisualizer3D.PoseCalibrationRoutine(vrRunning,
                poseTDimensionsCalculated => {
                    PoseVisualizer3D.ScalePose(vrTDimensions,
                        poseTDimensionsCalculated);
                    Debug.Log("VR calibration done!");
                    Instruction.SetActive(false);
                }));
        }
    }));
}

```



```

        }));
    }
}

private IEnumerator VrCalibrationRoutine(System.Action<Vector3>
    callback = null)
{
    yield return new WaitForSeconds(5);
    Vector3 vrTDimensions = Vector3.zero;
    vrTDimensions.x = Vector3.Distance(leftControllerPosition,
        rightControllerPosition);
    vrTDimensions.y = hmdPosition.y;
    ScaleAvatar(vrTDimensions);
    callback (vrTDimensions);
}

/// <summary>
/// Scale the avatar based on the physical dimensions of the user's body
/// </summary>
private void ScaleAvatar(Vector3 bodyTDimensions)
{
    Vector3 scaling;
    scaling.x = bodyTDimensions.x / avatarDimensions.x;
    scaling.y = bodyTDimensions.y / avatarDimensions.y;
    scaling.z = (scaling.x + scaling.y) / 2f;
    transform.localScale = scaling;
    jointPositionOffset.y = avatarCenter.y - avatarCenter.y * scaling.y;
    Debug.Log("Avatar scaling done");
}
}

```

A.2 PoseVisualizer3D.cs

```

using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using Mediapipe.BlazePose;
using System.Collections;
using System.Linq;

public class PoseVisualizer3D : MonoBehaviour
{

```

```

[SerializeField] Camera mainCamera;
[SerializeField] WebCamInput webCamInput;
[SerializeField] RawImage inputImageUI;
[SerializeField] Shader shader;
[SerializeField, Range(0, 1)] float humanExistThreshold = 0.5f;

Material material;
BlazePoseDetector detector;

// Lines count of body's topology.
const int BODY_LINE_NUM = 35;
// Pairs of vertex indices of the lines that make up body's topology.
// Defined by the figure in
    https://google.github.io/mediapipe/solutions/pose.
readonly List<Vector4> linePair = new List<Vector4>
{
    new Vector4(0, 1), new Vector4(1, 2), new Vector4(2, 3), new
        Vector4(3, 7), new Vector4(0, 4),
    new Vector4(4, 5), new Vector4(5, 6), new Vector4(6, 8), new
        Vector4(9, 10), new Vector4(11, 12),
    new Vector4(11, 13), new Vector4(13, 15), new Vector4(15, 17), new
        Vector4(17, 19), new Vector4(19, 15),
    new Vector4(15, 21), new Vector4(12, 14), new Vector4(14, 16), new
        Vector4(16, 18), new Vector4(18, 20),
    new Vector4(20, 16), new Vector4(16, 22), new Vector4(11, 23), new
        Vector4(12, 24), new Vector4(23, 24),
    new Vector4(23, 25), new Vector4(25, 27), new Vector4(27, 29), new
        Vector4(29, 31), new Vector4(31, 27),
    new Vector4(24, 26), new Vector4(26, 28), new Vector4(28, 30), new
        Vector4(30, 32), new Vector4(32, 28)
};

public VNectModel VNectModel;

/// <summary>
/// Coordinates of joint points
/// </summary>
private VNectModel.JointPoint[] jointPoints;
private Vector3 scaling = Vector3.one;
public bool showSkeleton;

void Start()
{
    material = new Material(shader);
    detector = new BlazePoseDetector();
    jointPoints = VNectModel.Initialize();

```

```

}

void Update()
{
    inputImageUI.texture = webCamInput.inputImageTexture;

    // Predict pose by neural network model.
    detector.ProcessImage(webCamInput.inputImageTexture);

    // Output landmark values(33 values) and the score whether human
    // pose is visible (1 values).
    for(int i = 0; i < detector.vertexCount + 1; i++)
    {
        /*
        0~32 index datas are pose world landmark.
        Check below Mediapipe document about relation between index and
        landmark position.
        https://google.github.io/mediapipe/solutions/pose#pose-landmark-model-blazepose-ghum-3d
        Each data factors are
        x, y and z: Real-world 3D coordinates in meters with the origin
        at the center between hips.
        w: The score of whether the world landmark position is visible
        ([0, 1]).

        33 index data is the score whether human pose is visible ([0,
        1]).
        This data is (score, 0, 0, 0).
        */
        // Debug.LogFormat("{0}: {1}", i,
        //     detector.GetPoseWorldLandmark(i));
        if (detector.GetPoseWorldLandmark(i).w > humanExistThreshold)
        {
            jointPoints[i].Pos3D.x = -1f *
                detector.GetPoseWorldLandmark(i).x * scaling.x;
            jointPoints[i].Pos3D.y = detector.GetPoseWorldLandmark(i).y *
                scaling.y;
            jointPoints[i].Pos3D.z = -1f *
                detector.GetPoseWorldLandmark(i).z * scaling.z;
            jointPoints[i].score3D = detector.GetPoseWorldLandmark(i).w;
        }
    }
    // Debug.Log("---");

    // Calculate head position
    Vector3 earCenter =
        Vector3.Lerp(jointPoints[PositionIndex.rEar.Int()].Pos3D,
            jointPoints[PositionIndex.lEar.Int()].Pos3D, 0.5f);

```

```

Vector3 eyeCenter =
    Vector3.Lerp(jointPoints[PositionIndex.rEye.Int()].Pos3D,
        jointPoints[PositionIndex.lEye.Int()].Pos3D, 0.5f);
Vector3 earCenterEyeCenter = eyeCenter - earCenter;
Vector3 leftEarRightEar =
    jointPoints[PositionIndex.rEar.Int()].Pos3D -
    jointPoints[PositionIndex.lEar.Int()].Pos3D;
Vector3 earCenterHead = Vector3.Cross(leftEarRightEar,
    earCenterEyeCenter);
Vector3 normalizedEarCenterHead = Vector3.Normalize(earCenterHead);
earCenterHead = normalizedEarCenterHead * 0.1f;
jointPoints[PositionIndex.head.Int()].Pos3D = earCenter +
    earCenterHead;
// Calculate head score
float[] headScores3D = {
    jointPoints[PositionIndex.rEar.Int()].score3D,
    jointPoints[PositionIndex.lEar.Int()].score3D,
    jointPoints[PositionIndex.rEye.Int()].score3D,
    jointPoints[PositionIndex.lEye.Int()].score3D };
jointPoints[PositionIndex.head.Int()].score3D = headScores3D.Min();

// Calculate neck position
Vector3 shoulderCenter =
    Vector3.Lerp(jointPoints[PositionIndex.rShoulder.Int()].Pos3D,
        jointPoints[PositionIndex.lShoulder.Int()].Pos3D, 0.5f);
jointPoints[PositionIndex.neck.Int()].Pos3D =
    Vector3.Lerp(shoulderCenter,
        jointPoints[PositionIndex.head.Int()].Pos3D, 0.3f);
// Calculate neck score
float[] neckScores3D = {
    jointPoints[PositionIndex.rShoulder.Int()].score3D,
    jointPoints[PositionIndex.lShoulder.Int()].score3D,
    jointPoints[PositionIndex.head.Int()].score3D };
jointPoints[PositionIndex.neck.Int()].score3D = neckScores3D.Min();

// Calculate hips position
Vector3 hipCenter =
    Vector3.Lerp(jointPoints[PositionIndex.rHip.Int()].Pos3D,
        jointPoints[PositionIndex.lHip.Int()].Pos3D, 0.5f);
jointPoints[PositionIndex.hips.Int()].Pos3D =
    Vector3.Lerp(hipCenter, shoulderCenter, 0.125f);
// Calculate hips score
float[] hipsScores3D = {
    jointPoints[PositionIndex.rShoulder.Int()].score3D,
    jointPoints[PositionIndex.lShoulder.Int()].score3D,

```

```

        jointPoints[PositionIndex.rHip.Int()].score3D,
        jointPoints[PositionIndex.lHip.Int()].score3D};
    jointPoints[PositionIndex.hips.Int()].score3D = hipsScores3D.Min();

    // Calculate spine position
    jointPoints[PositionIndex.spine.Int()].Pos3D =
        Vector3.Lerp(hipCenter, shoulderCenter, 0.28f);
    // Calculate spine score
    jointPoints[PositionIndex.spine.Int()].score3D = hipsScores3D.Min();

    // Calculate chest position
    jointPoints[PositionIndex.chest.Int()].Pos3D =
        Vector3.Lerp(hipCenter, shoulderCenter, 0.7f);
    // Calculate chest score
    jointPoints[PositionIndex.chest.Int()].score3D = hipsScores3D.Min();
}

void OnRenderObject()
{
    if(showSkeleton)
    {
        // Use predicted pose world landmark results on the
        // ComputeBuffer (GPU) memory.
        material.SetBuffer("_worldVertices",
            detector.worldLandmarkBuffer);
        // Set pose landmark counts.
        material.SetInt("_keypointCount", detector.vertexCount);
        material.SetFloat("_humanExistThreshold", humanExistThreshold);
        material.SetVectorArray("_linePair", linePair);
        material.SetMatrix("_invViewMatrix",
            mainCamera.worldToCameraMatrix.inverse);

        // Draw 35 world body topology lines.
        material.SetPass(2);
        Graphics.DrawProceduralNow(MeshTopology.Triangles, 6,
            BODY_LINE_NUM);

        // Draw 33 world landmark points.
        material.SetPass(3);
        Graphics.DrawProceduralNow(MeshTopology.Triangles, 6,
            detector.vertexCount);
    }
}

void OnApplicationQuit()
{
    // Must call Dispose method when no longer in use.

```

```

        detector.Dispose();
    }

    public IEnumerator PoseCalibrationRoutine(bool vrRunning,
        System.Action<Vector3> callback = null)
    {
        if (!vrRunning)
            yield return new WaitForSeconds(5);
        Vector3 poseTDimensions = Vector3.zero;
        poseTDimensions.x =
            Vector3.Distance(detector.GetPoseWorldLandmark(15),
                detector.GetPoseWorldLandmark(16));
        float floor = Mathf.Min(detector.GetPoseWorldLandmark(29).y,
            detector.GetPoseWorldLandmark(30).y,
            detector.GetPoseWorldLandmark(31).y,
            detector.GetPoseWorldLandmark(32).y);
        poseTDimensions.y = detector.GetPoseWorldLandmark(0).y - floor;
        callback (poseTDimensions);
    }

    /// <summary>
    /// Scale BlazePose based on the physical dimensions of the user's body
    /// measured using HMD and controllers
    /// </summary>
    public void ScalePose(Vector3 vrTDimensions, Vector3 poseTDimensions)
    {
        scaling.x = vrTDimensions.x / poseTDimensions.x;
        scaling.y = vrTDimensions.y / poseTDimensions.y;
        scaling.z = (scaling.x + scaling.y) / 2f;
        Debug.Log("BlazePose scaling done");
    }
}

```

A.3 BodySourceView.cs

```

using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using Kinect = Windows.Kinect;

public class BodySourceView : MonoBehaviour
{
    public Material BoneMaterial;
    public GameObject BodySourceManager;
}

```

```

public GameObject FaceManager;
public VNectModel VNectModel;

/// <summary>
/// Coordinates of joint points
/// </summary>
private VNectModel.JointPoint[] jointPoints;
private Vector3 spineMid;
private Vector3 footLeft;
private Vector3 footRight;
private Vector3 spineShoulder;
private Vector3 handLeft;
private Vector3 handRight;
private Vector3 handTipLeft;
private Vector3 handTipRight;
private Dictionary<ulong, GameObject> _Bodies = new Dictionary<ulong,
    GameObject>();
private BodySourceManager _BodyManager;
private FaceManager _FaceManager;
private Vector3[] kinectJoints = new Vector3[47]; // Kinect's joints
    data (with rotation, without scaling)
private Vector3 scaling = Vector3.one;
float kinectOffset = 0;
public bool showSkeleton;

private Dictionary<Kinect.JointType, Kinect.JointType> _BoneMap = new
    Dictionary<Kinect.JointType, Kinect.JointType>()
{
    { Kinect.JointType.FootLeft, Kinect.JointType.AnkleLeft },
    { Kinect.JointType.AnkleLeft, Kinect.JointType.KneeLeft },
    { Kinect.JointType.KneeLeft, Kinect.JointType.HipLeft },
    { Kinect.JointType.HipLeft, Kinect.JointType.SpineBase },

    { Kinect.JointType.FootRight, Kinect.JointType.AnkleRight },
    { Kinect.JointType.AnkleRight, Kinect.JointType.KneeRight },
    { Kinect.JointType.KneeRight, Kinect.JointType.HipRight },
    { Kinect.JointType.HipRight, Kinect.JointType.SpineBase },

    { Kinect.JointType.HandTipLeft, Kinect.JointType.HandLeft },
    { Kinect.JointType.ThumbLeft, Kinect.JointType.HandLeft },
    { Kinect.JointType.HandLeft, Kinect.JointType.WristLeft },
    { Kinect.JointType.WristLeft, Kinect.JointType.ElbowLeft },
    { Kinect.JointType.ElbowLeft, Kinect.JointType.ShoulderLeft },
    { Kinect.JointType.ShoulderLeft, Kinect.JointType.SpineShoulder },

    { Kinect.JointType.HandTipRight, Kinect.JointType.HandRight },
    { Kinect.JointType.ThumbRight, Kinect.JointType.HandRight },

```

```

{ Kinect.JointType.HandRight, Kinect.JointType.WristRight },
{ Kinect.JointType.WristRight, Kinect.JointType.ElbowRight },
{ Kinect.JointType.ElbowRight, Kinect.JointType.ShoulderRight },
{ Kinect.JointType.ShoulderRight, Kinect.JointType.SpineShoulder },

{ Kinect.JointType.SpineBase, Kinect.JointType.SpineMid },
{ Kinect.JointType.SpineMid, Kinect.JointType.SpineShoulder },
{ Kinect.JointType.SpineShoulder, Kinect.JointType.Neck },
{ Kinect.JointType.Neck, Kinect.JointType.Head },
};

void Start()
{
    jointPoints = VNectModel.Initialize();
}

void Update ()
{
    if (BodySourceManager == null)
    {
        return;
    }

    if (FaceManager == null)
    {
        return;
    }

    _BodyManager = BodySourceManager.GetComponent<BodySourceManager>();
    if (_BodyManager == null)
    {
        return;
    }

    _FaceManager = FaceManager.GetComponent<FaceManager>();
    if (_FaceManager == null)
    {
        return;
    }

    Kinect.Body[] data = _BodyManager.GetData();
    if (data == null)
    {
        return;
    }

    List<ulong> trackedIds = new List<ulong>();

```



```

foreach(var body in data)
{
    if (body == null)
    {
        continue;
    }

    if(body.IsTracked)
    {
        trackedIds.Add (body.TrackingId);
    }
}

List<ulong> knownIds = new List<ulong>(_Bodies.Keys);

// First delete untracked bodies
foreach(ulong trackingId in knownIds)
{
    if(!trackedIds.Contains(trackingId))
    {
        Destroy(_Bodies[trackingId]);
        _Bodies.Remove(trackingId);
    }
}

foreach(var body in data)
{
    if (body == null)
    {
        continue;
    }

    if(body.IsTracked)
    {
        if(!_Bodies.ContainsKey(body.TrackingId))
        {
            _Bodies[body.TrackingId] =
                CreateBodyObject(body.TrackingId);
        }

        RefreshBodyObject(body, _Bodies[body.TrackingId]);

        // Calculations of the missing joints
        jointPoints[PositionIndex.lFootIndex.Int()].Pos3D = footLeft;
        jointPoints[PositionIndex.rFootIndex.Int()].Pos3D = footRight;
        jointPoints[PositionIndex.neck.Int()].Pos3D =
            Vector3.Lerp(spineShoulder,

```

```

        jointPoints[PositionIndex.neck.Int()].Pos3D, 0.5f);
    Vector3 hipCenter =
        Vector3.Lerp(jointPoints[PositionIndex.rHip.Int()].Pos3D,
            jointPoints[PositionIndex.lHip.Int()].Pos3D, 0.5f);
    jointPoints[PositionIndex.hips.Int()].Pos3D =
        Vector3.Lerp(hipCenter, spineMid, 0.25f);
    jointPoints[PositionIndex.chest.Int()].Pos3D =
        Vector3.Lerp(spineMid, spineShoulder, 0.2f);
    jointPoints[PositionIndex.spine.Int()].Pos3D =
        Vector3.Lerp(jointPoints[PositionIndex.hips.Int()].Pos3D,
            spineMid, 0.3f);
    Vector3 lMiddleProximal = Vector3.Lerp(handLeft, handTipLeft,
        0.5f);
    Vector3 lThumbMiddleProximal = lMiddleProximal -
        jointPoints[PositionIndex.lThumb.Int()].Pos3D;
    jointPoints[PositionIndex.lPinky.Int()].Pos3D =
        jointPoints[PositionIndex.lThumb.Int()].Pos3D + 1.5f *
        lThumbMiddleProximal;
    Vector3 rMiddleProximal = Vector3.Lerp(handRight,
        handTipRight, 0.5f);
    Vector3 rThumbMiddleProximal = rMiddleProximal -
        jointPoints[PositionIndex.rThumb.Int()].Pos3D;
    jointPoints[PositionIndex.rPinky.Int()].Pos3D =
        jointPoints[PositionIndex.rThumb.Int()].Pos3D + 1.5f *
        rThumbMiddleProximal;
    }
}

// Face points
jointPoints[PositionIndex.Nose.Int()].Pos3D =
    _FaceManager.GetFacePointsPosition(0);
jointPoints[PositionIndex.lEye.Int()].Pos3D =
    _FaceManager.GetFacePointsPosition(1);
jointPoints[PositionIndex.rEye.Int()].Pos3D =
    _FaceManager.GetFacePointsPosition(2);
jointPoints[PositionIndex.lEar.Int()].Pos3D =
    _FaceManager.GetFacePointsPosition(3);
jointPoints[PositionIndex.rEar.Int()].Pos3D =
    _FaceManager.GetFacePointsPosition(4);
}

private GameObject CreateBodyObject(ulong id)
{
    GameObject body = new GameObject("Body:" + id);

    // Skeleton visualizer
    if (showSkeleton)

```

```

    {
        for (Kinect.JointType jt = Kinect.JointType.SpineBase; jt <=
            Kinect.JointType.ThumbRight; jt++)
        {
            GameObject jointObj =
                GameObject.CreatePrimitive(PrimitiveType.Cube);

            LineRenderer lr = jointObj.AddComponent<LineRenderer>();
            lr.positionCount = 2;
            lr.material = BoneMaterial;
            lr.startWidth = 0.05f;
            lr.endWidth = 0.05f;

            jointObj.transform.localScale = new Vector3(0.1f, 0.1f, 0.1f);
            jointObj.name = jt.ToString();
            jointObj.transform.parent = body.transform;
        }
    }

    return body;
}

private void RefreshBodyObject(Kinect.Body body, GameObject bodyObject)
{
    for (Kinect.JointType jt = Kinect.JointType.SpineBase; jt <=
        Kinect.JointType.ThumbRight; jt++)
    {
        Kinect.Joint sourceJoint = body.Joints[jt];
        Kinect.Joint? targetJoint = null;

        // Skeleton visualizer
        if (showSkeleton)
        {
            if (_BoneMap.ContainsKey(jt))
            {
                targetJoint = body.Joints[_BoneMap[jt]];
            }

            Transform jointObj = bodyObject.transform.Find(jt.ToString());
            jointObj.localPosition = new Vector3 (sourceJoint.Position.X,
                sourceJoint.Position.Y, (sourceJoint.Position.Z * -1f) +
                2f);

            LineRenderer lr = jointObj.GetComponent<LineRenderer>();
            if (targetJoint.HasValue)
            {

```

```

        lr.SetPosition(0, jointObj.localPosition);
        lr.SetPosition(1, new Vector3 (sourceJoint.Position.X,
            sourceJoint.Position.Y, (sourceJoint.Position.Z *
            -1f) + 2f));
        lr.startColor = GetColorForState
            (sourceJoint.TrackingState);
        lr.endColor =
            GetColorForState(targetJoint.Value.TrackingState);
    }
    else
    {
        lr.enabled = false;
    }
}

// Mapping of Kinect joints to VNectModel joints
if (jt.ToString().Equals("SpineMid"))
{
    spineMid = GetVector3FromJoint(sourceJoint,
        PositionIndex.spine.Int());
}

if (jt.ToString().Equals("Neck"))
{
    jointPoints[PositionIndex.head.Int()].Pos3D =
        GetVector3FromJoint(sourceJoint,
            PositionIndex.head.Int());
}

if (jt.ToString().Equals("Head"))
{
    jointPoints[PositionIndex.centerHead.Int()].Pos3D =
        GetVector3FromJoint(sourceJoint,
            PositionIndex.centerHead.Int());
}

if (jt.ToString().Equals("ShoulderLeft"))
{
    jointPoints[PositionIndex.lShoulder.Int()].Pos3D =
        GetVector3FromJoint(sourceJoint,
            PositionIndex.lShoulder.Int());
}

if (jt.ToString().Equals("ElbowLeft"))
{

```

```
        jointPoints[PositionIndex.lElbow.Int()].Pos3D =
            GetVector3FromJoint(sourceJoint,
                PositionIndex.lElbow.Int());
    }

    if (jt.ToString().Equals("WristLeft"))
    {
        jointPoints[PositionIndex.lWrist.Int()].Pos3D =
            GetVector3FromJoint(sourceJoint,
                PositionIndex.lWrist.Int());
    }

    if (jt.ToString().Equals("HandLeft"))
    {
        handLeft = GetVector3FromJoint(sourceJoint,
            PositionIndex.lController.Int());
    }

    if (jt.ToString().Equals("ShoulderRight"))
    {
        jointPoints[PositionIndex.rShoulder.Int()].Pos3D =
            GetVector3FromJoint(sourceJoint,
                PositionIndex.rShoulder.Int());
    }

    if (jt.ToString().Equals("ElbowRight"))
    {
        jointPoints[PositionIndex.rElbow.Int()].Pos3D =
            GetVector3FromJoint(sourceJoint,
                PositionIndex.rElbow.Int());
    }

    if (jt.ToString().Equals("WristRight"))
    {
        jointPoints[PositionIndex.rWrist.Int()].Pos3D =
            GetVector3FromJoint(sourceJoint,
                PositionIndex.rWrist.Int());
    }

    if (jt.ToString().Equals("HandRight"))
    {
        handRight = GetVector3FromJoint(sourceJoint,
            PositionIndex.rController.Int());
    }

    if (jt.ToString().Equals("HipLeft"))
    {
```

```

        jointPoints[PositionIndex.lHip.Int()].Pos3D =
            GetVector3FromJoint(sourceJoint,
                PositionIndex.lHip.Int());
    }

    if (jt.ToString().Equals("KneeLeft"))
    {
        jointPoints[PositionIndex.lKnee.Int()].Pos3D =
            GetVector3FromJoint(sourceJoint,
                PositionIndex.lKnee.Int());
    }

    if (jt.ToString().Equals("AnkleLeft"))
    {
        jointPoints[PositionIndex.lAnkle.Int()].Pos3D =
            GetVector3FromJoint(sourceJoint,
                PositionIndex.lAnkle.Int());
    }

    if (jt.ToString().Equals("FootLeft"))
    {
        footLeft = GetVector3FromJoint(sourceJoint,
            PositionIndex.lFootIndex.Int());
    }

    if (jt.ToString().Equals("HipRight"))
    {
        jointPoints[PositionIndex.rHip.Int()].Pos3D =
            GetVector3FromJoint(sourceJoint,
                PositionIndex.rHip.Int());
    }

    if (jt.ToString().Equals("KneeRight"))
    {
        jointPoints[PositionIndex.rKnee.Int()].Pos3D =
            GetVector3FromJoint(sourceJoint,
                PositionIndex.rKnee.Int());
    }

    if (jt.ToString().Equals("AnkleRight"))
    {
        jointPoints[PositionIndex.rAnkle.Int()].Pos3D =
            GetVector3FromJoint(sourceJoint,
                PositionIndex.rAnkle.Int());
    }

    if (jt.ToString().Equals("FootRight"))

```

```

    {
        footRight = GetVector3FromJoint(sourceJoint,
            PositionIndex.rFootIndex.Int());
    }

    if (jt.ToString().Equals("SpineShoulder"))
    {
        spineShoulder = GetVector3FromJoint(sourceJoint,
            PositionIndex.chest.Int());
    }

    if (jt.ToString().Equals("HandTipLeft"))
    {
        handTipLeft = GetVector3FromJoint(sourceJoint,
            PositionIndex.lIndex.Int());
    }

    if (jt.ToString().Equals("ThumbLeft"))
    {
        jointPoints[PositionIndex.lThumb.Int()].Pos3D =
            GetVector3FromJoint(sourceJoint,
                PositionIndex.lThumb.Int());
    }

    if (jt.ToString().Equals("HandTipRight"))
    {
        handTipRight = GetVector3FromJoint(sourceJoint,
            PositionIndex.rIndex.Int());
    }

    if (jt.ToString().Equals("ThumbRight"))
    {
        jointPoints[PositionIndex.rThumb.Int()].Pos3D =
            GetVector3FromJoint(sourceJoint,
                PositionIndex.rThumb.Int());
    }
    }
}

private static Color GetColorForState(Kinect.TrackingState state)
{
    switch (state)
    {
        case Kinect.TrackingState.Tracked:
            return Color.green;

        case Kinect.TrackingState.Inferred:

```

```

        return Color.red;

    default:
        return Color.black;
    }
}

private Vector3 GetVector3FromJoint(Kinect.Joint joint, int jointNumber)
{
    kinectJoints[jointNumber] = new Vector3(joint.Position.X,
        joint.Position.Y, (joint.Position.Z * -1f) + 2f);
    return Vector3.Scale(kinectJoints[jointNumber], scaling) - new
        Vector3(0f, kinectOffset, 0f);
}

public IEnumerator KinectCalibrationRoutine(bool vrRunning,
    System.Action<Vector3> callback = null)
{
    if (!vrRunning)
        yield return new WaitForSeconds(5);
    Vector3 kinectTDimensions = Vector3.zero;
    kinectTDimensions.x =
        Vector3.Distance(kinectJoints[PositionIndex.lController.Int()],
            kinectJoints[PositionIndex.rController.Int()]);
    Vector3 hipCenter =
        Vector3.Lerp(kinectJoints[PositionIndex.rHip.Int()],
            kinectJoints[PositionIndex.lHip.Int()], 0.5f);
    Vector3 hips = Vector3.Lerp(hipCenter,
        kinectJoints[PositionIndex.spine.Int()], 0.25f);
    float floor =
        Mathf.Min(kinectJoints[PositionIndex.lFootIndex.Int()].y,
            kinectJoints[PositionIndex.rFootIndex.Int()].y);
    if (vrRunning)
        kinectOffset = -floor + hips.y;
    kinectTDimensions.y = kinectJoints[PositionIndex.centerHead.Int()].y
        - floor;
    callback (kinectTDimensions);
}

/// <summary>
/// Scale Kinect based on the physical dimensions of the user's body
/// measured using HMD and controllers
/// </summary>
public void ScaleKinect(Vector3 vrTDimensions, Vector3
    kinectTDimensions)
{
    scaling.x = vrTDimensions.x / kinectTDimensions.x;
}

```



```

        scaling.y = vrTDimensions.y / kinectTDimensions.y;
        scaling.z = (scaling.x + scaling.y) / 2f;
        Debug.Log("Kinect scaling done");
    }
}

```

A.4 FaceManager.cs

```

using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using Windows.Kinect;
using Microsoft.Kinect.Face;
using System.Linq;

public static partial class EnumExtend
{
    public static int Int(this HighDetailFacePoints i)
    {
        return (int)i;
    }
}

public class FaceManager : MonoBehaviour
{
    private KinectSensor kinectSensor;
    private int bodyCount;
    private Body[] bodies;
    private BodySourceManager bodySourceManager;
    private FaceFrameSource[] faceFrameSources;
    private FaceFrameReader[] faceFrameReaders;
    private BodyFrameSource _bodySource = null;
    private BodyFrameReader _bodyReader = null;
    private HighDefinitionFaceFrameSource _hdFaceFrameSources = null;
    private HighDefinitionFaceFrameReader _hdFaceFrameReaders = null;
    public BodySourceManager bodyManager;
    private FaceAlignment _faceAlignment = null;
    private FaceModel _faceModel = null;
    private Quaternion faceRotation;
    private Vector3 leftEyeMidTop;
    private Vector3 leftEyeMidBottom;
    private Vector3 rightEyeMidTop;
    private Vector3 rightEyeMidBottom;
    private Vector3 leftCheekCenter;

```

```

private Vector3 rightCheekCenter;
private Vector3 leftCheekBone;
private Vector3 rightCheekBone;
private Vector3[] facePointsPosition = new Vector3[5];

void Start()
{
    // one sensor is currently supported
    kinectSensor = KinectSensor.GetDefault();

    // set the maximum number of bodies that would be tracked by Kinect
    bodyCount = kinectSensor.BodyFrameSource.BodyCount;

    // allocate storage to store body objects
    bodies = new Body[bodyCount];

    // get bodies either from BodySourceManager object get them from a
    BodyReader
    bodySourceManager = bodyManager.GetComponent<BodySourceManager>();

    // specify the required face frame results
    FaceFrameFeatures faceFrameFeatures =
        FaceFrameFeatures.BoundingBoxInColorSpace
        | FaceFrameFeatures.PointsInColorSpace
        | FaceFrameFeatures.BoundingBoxInInfraredSpace
        | FaceFrameFeatures.PointsInInfraredSpace
        | FaceFrameFeatures.RotationOrientation
        | FaceFrameFeatures.FaceEngagement
        | FaceFrameFeatures.Glasses
        | FaceFrameFeatures.Happy
        | FaceFrameFeatures.LeftEyeClosed
        | FaceFrameFeatures.RightEyeClosed
        | FaceFrameFeatures.LookingAway
        | FaceFrameFeatures.MouthMoved
        | FaceFrameFeatures.MouthOpen;

    // create a face frame source + reader to track each face in the FOV
    faceFrameSources = new FaceFrameSource[bodyCount];
    faceFrameReaders = new FaceFrameReader[bodyCount];
    for (int i = 0; i < bodyCount; i++)
    {
        // create the face frame source with the required face frame
        features and an initial tracking Id of 0
        faceFrameSources[i] = FaceFrameSource.Create(kinectSensor, 0,
            faceFrameFeatures);

        // open the corresponding reader
    }
}

```

```

        faceFrameReaders[i] = faceFrameSources[i].OpenReader();
    }

    _bodySource = kinectSensor.BodyFrameSource;
    _bodyReader = _bodySource.OpenReader();
    _bodyReader.FrameArrived += BodyReader_FrameArrived;

    _hdFaceFrameSources =
        HighDefinitionFaceFrameSource.Create(kinectSensor);

    _hdFaceFrameReaders = _hdFaceFrameSources.OpenReader();
    _hdFaceFrameReaders.FrameArrived += FaceReader_FrameArrived;

    _faceModel = FaceModel.Create();
    _faceAlignment = FaceAlignment.Create();
}

void Update()
{
    bodies = bodySourceManager.GetData();

    if (bodies == null)
    {
        return;
    }

    // iterate through each body and update face source
    for (int i = 0; i < bodyCount; i++)
    {
        // check if a valid face is tracked in this face source
        if (faceFrameSources[i].IsTrackingIdValid)
        {
            using (FaceFrame frame = faceFrameReaders[i].AcquireLatestFrame())
            {
                if (frame != null)
                {
                    if (frame.TrackingId == 0)
                    {
                        continue;
                    }

                    // do something with the result
                    var result = frame.FaceFrameResult;

                    var faceRotationVector4 = result.FaceRotationQuaternion;
                    faceRotation.Set(faceRotationVector4.X,
                                    faceRotationVector4.Y, faceRotationVector4.Z,

```

```

        faceRotationVector4.W);
    }
}
else
{
    if (bodies[i] == null)
        return;

    // check if the corresponding body is tracked
    if (bodies[i].IsTracked)
    {
        // update the face frame source to track this body
        faceFrameSources[i].TrackingId = bodies[i].TrackingId;
    }
}
}

    RetriveFacePoints();
}

public Quaternion GetFaceRotation()
{
    return faceRotation;
}

private void BodyReader_FrameArrived(object sender,
    BodyFrameArrivedEventArgs e)
{
    using (var frame = e.FrameReference.AcquireFrame())
    {
        if (frame != null)
        {
            Body[] bodies = new Body[frame.BodyCount];
            frame.GetAndRefreshBodyData(bodies);

            Body body = bodies.Where(b => b.IsTracked).FirstOrDefault();

            if (!_hdFaceFrameSources.IsTrackingIdValid)
            {
                if (body != null)
                {
                    _hdFaceFrameSources.TrackingId = body.TrackingId;
                }
            }
        }
    }
}

```

```

    }

    private void FaceReader_FrameArrived(object sender,
        HighDefinitionFaceFrameArrivedEventArgs e)
    {
        using (var frame = e.FrameReference.AcquireFrame())
        {
            if (frame != null && frame.IsFaceTracked)
            {
                frame.GetAndRefreshFaceAlignmentResult(_faceAlignment);
            }
        }
    }

    private void RetriveFacePoints()
    {
        if (_faceModel == null) return;

        var vertices =
            _faceModel.CalculateVerticesForAlignment(_faceAlignment);

        if (vertices.Count > 0)
        {
            for (int index = 0; index < vertices.Count; index++)
            {
                // Get face points positions

                if (index == HighDetailFacePoints.NoseTip.Int())
                    facePointsPosition[0] = GetVerticePosition(vertices[index]);

                if (index == HighDetailFacePoints.LefteyeMidtop.Int())
                    leftEyeMidTop = GetVerticePosition(vertices[index]);

                if (index == HighDetailFacePoints.LefteyeMidbottom.Int())
                    leftEyeMidBottom = GetVerticePosition(vertices[index]);

                facePointsPosition[1] = Vector3.Lerp(leftEyeMidTop,
                    leftEyeMidBottom, 0.5f);

                if (index == HighDetailFacePoints.RighteyeMidtop.Int())
                    rightEyeMidTop = GetVerticePosition(vertices[index]);

                if (index == HighDetailFacePoints.RighteyeMidbottom.Int())
                    rightEyeMidBottom = GetVerticePosition(vertices[index]);

                facePointsPosition[2] = Vector3.Lerp(rightEyeMidTop,
                    rightEyeMidBottom, 0.5f);
            }
        }
    }

```

```

        if(index == HighDetailFacePoints.LeftcheekCenter.Int())
            leftCheekCenter = GetVerticePosition(vertices[index]);

        if(index == HighDetailFacePoints.Leftcheekbone.Int())
            leftCheekBone = GetVerticePosition(vertices[index]);

        Vector3 leftCheek = Vector3.Lerp(leftCheekCenter, leftCheekBone,
            0.5f);
        Vector3 noseLeftCheek = leftCheek - facePointsPosition[0];
        facePointsPosition[3] = leftCheek + noseLeftCheek;

        if(index == HighDetailFacePoints.RightcheekCenter.Int())
            rightCheekCenter = GetVerticePosition(vertices[index]);

        if(index == HighDetailFacePoints.Rightcheekbone.Int())
            rightCheekBone = GetVerticePosition(vertices[index]);

        Vector3 rightCheek = Vector3.Lerp(rightCheekCenter,
            rightCheekBone, 0.5f);
        Vector3 noseRightCheek = rightCheek - facePointsPosition[0];
        facePointsPosition[4] = rightCheek + noseRightCheek;
    }
}

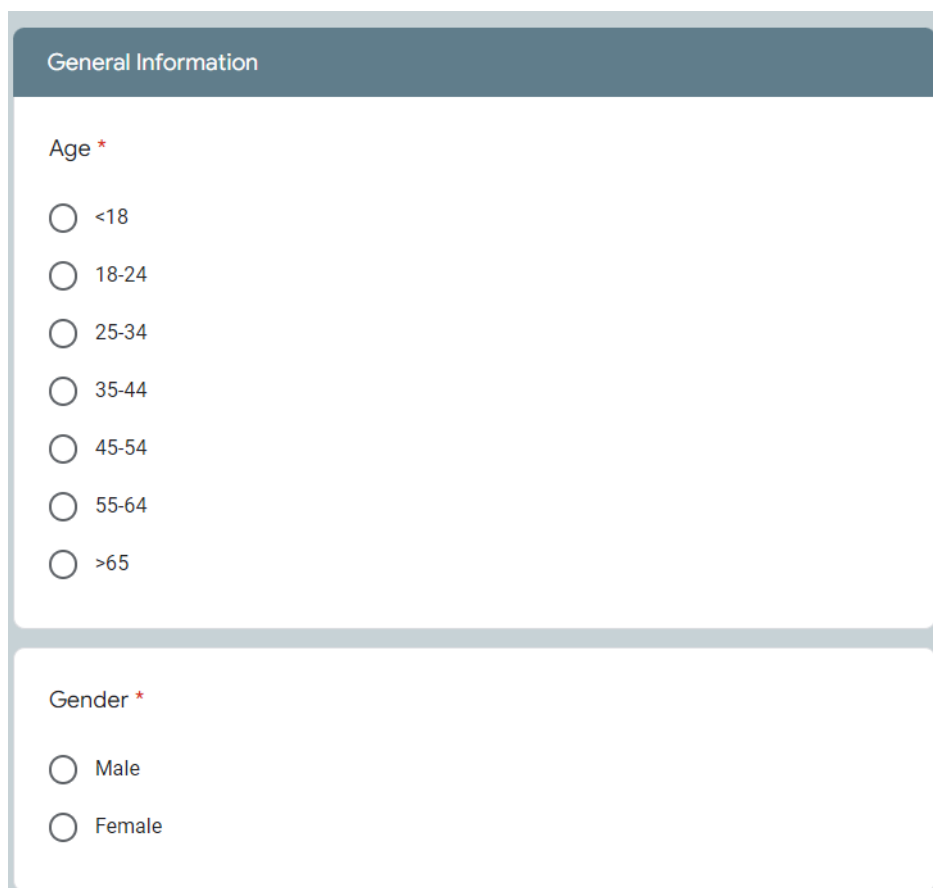
private Vector3 GetVerticePosition(Windows.Kinect.CameraSpacePoint
    vertice)
{
    return new Vector3(vertice.X, vertice.Y, (vertice.Z * -1f) + 2f);
}

public Vector3 GetFacePointsPosition(int index)
{
    return facePointsPosition[index];
}
}

```

Appendix B

Questionnaire



The image shows a digital questionnaire form with a dark blue header bar containing the text 'General Information'. Below the header, the form is divided into two sections. The first section is titled 'Age *' and contains seven radio button options: '<18', '18-24', '25-34', '35-44', '45-54', '55-64', and '>65'. The second section is titled 'Gender *' and contains two radio button options: 'Male' and 'Female'. The form has a light gray border and rounded corners.

General Information

Age *

☐ <18

☐ 18-24

☐ 25-34

☐ 35-44

☐ 45-54

☐ 55-64

☐ >65

Gender *

☐ Male

☐ Female

Figure B.1: Questionnaire part1:General Information

K-implementation

Do you believe that the avatar was accurately representing your body movements? *

1 2 3 4 5 6 7

Strongly disagree ☐ ☐ ☐ ☐ ☐ ☐ ☐ Strongly agree

If not, which parts of your body you feel like were less accurately represented.

Your answer _____

Did you notice any latency in the avatar's movement while you were performing the warm-up routine? *

1 2 3 4 5 6 7

Strongly disagree ☐ ☐ ☐ ☐ ☐ ☐ ☐ Strongly agree

If yes, which avatar body parts did you feel were most affected?

Your answer _____

Where there any body parts that were glitching during the test? *

1 2 3 4 5 6 7

Strongly disagree ○ ○ ○ ○ ○ ○ ○ Strongly agree

If yes, which parts of the avatars body were affected.

Your answer _____

Which movement direction was represented accurately? (you can choose neither, one or more than one)

☐ Right and Left movement

☐ Back and forth movement

☐ Up and down movement

Did you feel that movements of both upper and lower part of your body were represented accurately by the avatar? *

☐ Upper body was represented more accurately

☐ Lower body was represented more accurately

☐ Both were accurately represented

How did you perceive the overall experience in sense of being in the virtual environment? *

1 2 3 4 5 6 7

The experience felt artificial ○ ○ ○ ○ ○ ○ ○ The experience felt realistic

Figure B.2: Questionnaire part2:Kinect Implementation

B-implementation

Do you believe that the avatar was accurately representing your body movements? *

1234567

Strongly disagreeStrongly agree

If not, which parts of your body you feel like were less accurately represented.

Your answer

Did you notice any latency in the avatar's movement while you were performing the warm-up routine? *

1234567

Strongly disagreeStrongly agree

If yes, which avatar body parts did you feel were most affected?

Your answer

Where there any body parts that were glitching during the test? *

1234567

Strongly disagreeStrongly agree

If yes, which parts of the avatars body were affected.

Your answer _____

Which movement direction was represented accurately? (you can choose neither, one or more than one)

☐ Right and Left movement

☐ Back and forth movement

☐ Up and down movement

Did you feel that movements of both upper and lower part of your body were represented accurately by the avatar? *

☐ Upper body was represented more accurately

☐ Lower body was represented more accurately

☐ Both were accurately represented

How did you perceive the overall experience in sense of being in the virtual environment? *

1 2 3 4 5 6 7

The experience felt artificial ☐ ☐ ☐ ☐ ☐ ☐ ☐ The experience felt realistic

Figure B.3: Questionnaire part3:BlazePose implementation

Comparison

In your opinion, which implementation was more accurately representing your body movements? *

1 2 3 4 5 6 7

K-implementation ☐ ☐ ☐ ☐ ☐ ☐ ☐ B-implementation

In your opinion, which implementation had less latency in the avatar's movement while you were performing the warm-up routine? *

1 2 3 4 5 6 7

K-implementation ☐ ☐ ☐ ☐ ☐ ☐ ☐ B-implementation

In your opinion, which implementation had less glitching incidents in the avatar's movement? *

1 2 3 4 5 6 7

K-implementation ☐ ☐ ☐ ☐ ☐ ☐ ☐ B-implementation

In your opinion, which one of the implementations felt more realistic, if any? *

☐ K-implementation

☐ B-implementation

☐ Neither

If neither felt realistic, please note why.

Η απάντησή σας _____

Figure B.4: Questionnaire part4: Kinect-BlazePose comparison