

Master's Thesis

Volumetric Capture with multiple azure Kinects in Unreal Engine

Marios Charalambous

UNIVERSITY OF CYPRUS



DEPARTMENT OF COMPUTER SCIENCE

JUNE 2023

Volumetric Capture with multiple azure Kinects in Unreal Engine

Marios Charalambous
University of Cyprus, 2023

A Thesis
Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science
at the
University of Cyprus

Recommended for Acceptance
By the Department of Computer Science
June, 2023

APPROVAL PAGE

Master of Science Thesis

VOLUMETRIC CAPTURE WITH MULTIPLE AZURE KINECTS IN UNREAL ENGINE

Presented by
Marios Charalambous

Research Supervisor Andreas Aristidou

Committee Member Panayiotis Charalambous

Committee Member Yiorgos Chrysanthou

University of Cyprus

June 2023

ACKNOWLEDGEMENTS

This thesis is a diploma thesis in the context of the Master of Science in Computer Science program of the Faculty of Science of the University of Cyprus.

With the completion of my thesis, I would like to thank some of the people I met, collaborated with, and played a very important role in its realization.

I would like to thank my thesis supervisor, Dr. Andreas Aristidou, who entrusted me with this thesis and guided me in its execution.

I also thank Dr.. Panagiotis Charalambous who is the head of the V-EUPNEA MRG team of the CYENS Center of Excellence for the guidelines for the completion of the thesis and his continuous response to any issues that arose and the help he provided me to solve them.

ABSTRACT

Volumetric capture (Volumetric video) is a technique that allows to create “holographic” recordings of actors, sets and props. The technique can be used to create immersive stories that sometimes reflect aspects of reality better than realistic 3D models. For example, volumetric captures of actors do not seem to cause uncanny valley effect. Volumetric video is a core underlying technology for emerging Mixed Reality systems. What was previously available for a glimpse only in science fiction movies and futuristic predictions, now with the ongoing research on volumetric video capturing, coding and presentation, realistic mixed reality experiences are close to become a reality. At the same time, computer generated holography and other digital 3D projection techniques start to become more common and affordable. The emergence of solutions for capturing volumetric video and devices which can display volumetric video mixed with the real world are paving the way to a new media, where a real object and its volumetric virtual image are indistinguishable. Virtual simulation of human faces and facial movements has challenged media artists and computer scientists since the first realistic 3D renderings of a human face by Fred Parke in 1972. Today, a range of software and techniques are available for modelling virtual characters and their facial behavior in immersive environments, such as computer games or storyworlds. However, applying these techniques often requires large teams with multidisciplinary expertise, extensive amount of manual labour, as well as financial conditions that are not typically available for individual media artists.

In this thesis first created a metahuman from photographs with the help of KeenTools FaceBuilder and the MetaHuman plugin. Also thesis aims to use three Azure Kinects DK devices(depth sensors) to capture in real-time the "hologram" of the user. All of the above is built on Epic Games' Unreal Engine game platform in C++ programming language in visual studio 2022.

Content

Chapter 1	Introduction.....	1
	1.1 Introduction	1
	1.2 Motivation	1
	1.3 Related Work	2
	1.4 Thesis Organization	2
Chapter 2	Theoretical Background and Scene Description.....	4
	2.1 Theoretical Background	4
	2.1.1 Unreal Engine	4
	2.1.2 Microsoft Azure Kinect DK	5
	2.1.3 PCL Library C++	5
	2.1.4 Azure Kinect SDK	6
	2.1.5 Python Open3D	6
	2.2 Scene Description	6
	2.2.1 Scene 1 – The Metahuman Character of myself	6
	2.2.2 Scene 2 – Realtime Volumetric Capture in UE	7
Chapter 3	Methodology.....	19
	3.1 Process to create Metahuman Character of myself	19
	3.2 General overview how to create volumetric character with multiple depth sensors	10
	3.3 Setting up of Microsoft Azure kinects depth sensors	24
	3.4 Capturing the character – object	26
	3.5 Merging point cloud - Calibration	30
	3.6 Processing the data	32
	3.7 Animating the character	35
Chapter 4	Operations.....	36
	4.1 Should Register Multiple Kinects	
	4.2 Save Registration	

4.3	Register to World	
4.4	Record Point Cloud	
Chapter 5	Manual	
5.1	Scene 1 – The Metahuman Character of myself	
5.2	Scene 2 – Realtime Volumetric Capture in UE	
Chapter 6	Conclusions – Results and Future Work	
6.1	Conclusions – Results	46
6.1.1	Summary of the thesis	46
6.1.2	Discussion of the results	46
6.1.2.1	Chessboard vs Lattice	47
6.1.2.2	Calibration Mean Error for the positions of the depth sensors	49
6.1.2.3	Calibration Mean Error for lighting conditions.	53
6.1.2.4	Time needed for Calibration - lighting conditions.	54
6.1.2.5	Time and Calibration Mean Error – Change height	54
6.1.2.6	NFOV vs WFOV	55
6.1.2.7	Ball-Pivoting vs Poisson Algorithm	58
6.2	Future Work	61
7	Bibliography	62
8	Appendix	62
8.1	Source Code	62
8.2	Structure	63

List Figures

Figure 1 - Metahuman Character of myself (front)	19 -
Figure 2 - Metahuman Character of myself (right angle)	19 -
Figure 3 - Metahuman Character Control Rig – Up Arm.....	20 -
Figure 4 - Metahuman Character Control Rig – Up Arm.....	20 -
Figure 5 - Metahuman Character Control Rig – Face.....	20 -
Figure 6 - Initialization of the three azure kinect.....	21 -
Figure 7 - Start Capturing.....	21 -
Figure 8 - high-quality photo(front)	22 -
Figure 9 - high-quality photo(left angle)	22 -
Figure 10 - high-quality photo(right side)	22 -
Figure 11 - align face - left.....	23 -
Figure 12 - align face - up.....	23 -
Figure 13 - align face - left.....	23 -
Figure 14 - align face - front.....	23 -
Figure 15 - Kentool Mesh.....	24 -
Figure 16 - Kentool mesh with texture	24 -
Figure 17 - Kentool texture	24 -
Figure 18 - Align Face to create Metahuman.....	25 -
Figure 19 - Final texture.....	26 -
Figure 20 - Metahuman of myself.....	26 -
Figure 21 - NFOV distances	28 -
Figure 22 - WFOV distances	28 -
Figure 23 -Ideal configuration for triple azure kinects sensors(WFOV).....	29 -
Figure 24 - Ideal configuration for triple azure kinects sensors(NFOV).....	29 -
Figure 25 - Kinects Setup.....	29 -
Figure 26 - Point Cloud rendering in unreal engine.....	33 -
Figure 27 - Chessboard pattern	33 -
Figure 28 - Configuration for chessboard pattern	34 -
Figure 29 - lattice	35 -
Figure 30 - Merging Point cloud.....	36 -
Figure 31 - Record file	38 -
Figure 32 - Ball Bivoting Mesh - frame 2.....	39 -
Figure 33 - Ball Pivoting mesh – frame 1	39 -
Figure 34- Possion Reconstruction - frame 2.....	40 -
Figure 35 - Poisson Recostruction - frame 1	40 -
Figure 36 - frame 2 animation	41 -
Figure 37 - frame 1 animation	41 -
Figure 38 - frame 4 animation	41 -
Figure 39 - frame 3 animation	41 -
Figure 40 - Registration file.....	42 -
Figure 41 - Record File	43 -
Figure 42 - Animation Sequence.....	44 -
Figure 43 - Operations	45 -
Figure 44 - NFOV vs WFOV.....	55 -

Figure 45 - Point Cloud..... - 58 -

Figure 46 - Mesh radius=2 max_nn=30..... - 58 -

Figure 47 - Mesh radius=10 ,max_nn=300..... - 59 -

Figure 48 - Mesh radius=5, max_nn=90..... - 59 -

Figure 49 - mesh depth = 8 - 60 -

Figure 50 - mesh depth = 10 - 60 -

Figure 51 - mesh depth = 6 - 60 -

List Graphs

Graph 1 - Chessboard vs Lattice pattern (Mean Error)..... - 47 -

Graph 2 - Chessboard vs Lattice (Time needed for calibration) - 49 -

Graph 3 - Mean error for positions of the depth sensors..... - 50 -

Graph 4 - time needed for calibration the depth sensors - 51 -

Graph 5 - mean error for lighting conditions..... - 52 -

Graph 6 - time needed for calibration (lighting conditions) - 53 -

Graph 7 - NFOV vs WFOV Calibration Mean Error..... - 56 -

Graph 8 - NFOV vs WFOV Time for Calibration..... - 57 -

Chapter 1

1.1 Introduction

Volumetric capture technology has emerged as a powerful tool in the field of computer graphics and multimedia applications. This technology allows to create 3D models of real life objects and people, which are captured from multiple viewing angles in a given space. Various applications, such as video games, virtual reality, AR and teleconferencing among others, can benefit from these 3D models.

Volumetric capture technology involves capturing the geometry, texture and motion of an object in 3D space. This requires multiple cameras, depth sensors and various devices that collect images of the object from a number of angles and distances. Once the data has been collected, it is then processed to create a 3D model that can be viewed and manipulated in a real time.

Over the past years, researchers and practitioners have given a lot of attention to volumetric capture because they are trying different applications and techniques for improving its quality and efficiency. This has led to the development of new algorithms, hardware and software tools, which have made volumetric capture more accessible and affordable.

This thesis aims to use three Azure Kinects DK devices to capture in real-time the "hologram" of the user within the unreal engine which is a game creation platform. Further analysis of the process, difficulties, and results will be done later.

1.2 Motivation

With the passing of the years, we are getting closer and closer to the era of the "metaverse". This gives a huge motivation to someone to deal with volumetric capture because, with the results it has, the creation of realistic characters and objects, the experience in virtual and augmented reality become impressive. Thus, the main objective of this thesis was to create a realistic character, control various parameters and extract influencing results (such as camera position, lighting in the space, etc.) that would help in future projects.

1.3 Related Work

Continuing in this chapter, a brief summary of the relevant work done previously is provided. A large body of studies have been accumulated in the field of realistic human modelling and animation over the years. However, making use of a captured workflow based on automation and algorithms instead of traditional tridimensional modelling pipelines, is a relatively recent practice. This work is highly influenced by the achievements of the USC Institute for Creative Technologies Especially Digital Emily (Alexander, 2010) and Digital Ira (von der Pahlen et al., 2014) can be mentioned. as works, where high-end hardware and software were created in order to achieve a realistic virtual human being,. In our case, we seek for the same outcome but utilizing low-end and off-the-shelf solutions available for individual media artists. The work of Zollhöfer et al. (2018) also shares similar aspects with our workflow. In their article “State of the Art on Monocular 3D Face Reconstruction, Tracking, and Applications” they evaluate different algorithms for the capture and reconstruction of human faces, introducing also possibilities of capturing actors’ movements. With a strict engineering focus, their work communicates very well with the automated approach proposed in this thesis. Our work relates to the extended work of Mark Sagar et al. (2016) at the Laboratory for Animate Technologies of Auckland University , especially their work conducted on the ways the viewer may interact with virtual characters. The same is true also with facial game technologies, especially volumetric technologies used in games like L.A. Noire (Star, 2011) and the achievements of Ninja Theory’s Hellblade (2017). From the scientific view-point , the research by Mark Cavazza and others (2002) shares similarities with our work, particularly related to character based interactive storytelling. This work also seeks to relate the researches and developments in sequenced volumetric capture solutions, especially the experiments from the volumetric capture community, such as the work of Or and Anlen (2018), Dou et al. (2017), Scatter (2017), as well as volumetric capture studios as Microsoft Mixed Reality Capture Studio (2018) and Fraun-hofer Institute’s Volucap Studio (2018). The uncanny valley discussions, specially the psychological research by Jari Kätsyri et al. (2015), Aline W. de Borst and Beatrice de Gelder (2015) and Rachel McDonnell et al. (2012) are also of great value to this work, bringing and opening discussions on the way humans perceive and interact with virtual characters. Then, due to the rapid development of technology and the performance of computers in recent years, the results of volumetric capture systems are astonishing. In 2018 Zollhöfer published the paper "Live Volumetric Performance Capture" where presents a system for live volumetric performance capture, which can be used to create real-time 3D content. Also, "Volumetric

Capture of Humans with a Single RGBD Sensor" by J. Xie et al., IEEE Transactions on Visualization and Computer Graphics, 2019. This paper presents a method for volumetric capture of humans using a single RGBD sensor. Finally, another interesting paper is "Efficient Volumetric Video Coding for Interactive Applications" by H. Aksay et al., ACM Transactions on Multimedia Computing, Communications, and Applications, 2020. This paper presents an efficient method for compressing and transmitting volumetric video data, which is important for interactive applications.

1.4 Thesis Organization

Chapter 1 gives a general introduction to what volumetric capture is and what it can be used for. Also, the motivation and objectives of the thesis, creating a realistic virtual character and extracting results and finally the literature review, i.e. the works created before my own thesis. The rest of the report will follow the following structure:

Chapter 2 will present the system requirements covering both hardware and software, the packages used to carry out this work. In addition, the two scenes made on the unreal engine game platform will be presented and the way it works will be analyzed.

In chapter 3 the whole process of creating the virtual metahuman character and the process of creating a real-time volumetric capture character will be presented and analyzed.

In chapter 4 all the game functions will be extensively reported and analyzed.

Chapter 5 will present the user manual of the simulation game, i.e. how the user can run and see the results on the Unreal Engine platform.

In chapter 6 all results and conclusions will be analysed with graphical extraction and comparison of the methods. And finally the future research that can be done to improve the existing thesis.

Chapter 2

2 Theoretical Background – Frameworks - Tools and Scene Description

2.1 Theoretical Background – Frameworks - Tools	12
2.1.1 Unreal Engine	12
2.1.2 Microsoft Azure Kinect DK	13
2.1.3 PCL Library C++	13
2.1.4 Azure Kinect SDK	14
2.1.5 Python Open3D Library	14
2.1.6 OpenCV library	15
2.2 Challenges	16
2.2.1 Storage and Compression	16
2.2.2 Data Acquisition and Processing	16
2.2.3 Real-Time Rendering	17
2.2.4 Hardware and Sensor Limitations	17
2.2.5 Depth and Occlusion Challenges	17
2.3 Scene Description	18
2.3.1 Scene 1 – The Metahuman Character of myself	18
2.3.2 Scene 2 – Realtime Volumetric Capture in Unreal Engine	20

Before the analysis of the simulation, general information and concepts that are necessary to understand for the continuation of the work will be explained.

2.1 Theoretical Background

2.1.1 Unreal Engine

Unreal Engine **Σφάλμα! Το αρχείο προέλευσης της αναφοράς δεν βρέθηκε.** is a popular and powerful game engine developed and maintained by Epic Games. It is a collection of software tools and technologies that can be used to create high-quality, interactive 3D games, simulations, and other interactive applications.

Unreal Engine provides a wide range of features, including a visual scripting system called Blueprints, a robust physics engine, advanced AI systems, and support for virtual reality and augmented reality. It also supports a variety of platforms, including Windows, Mac, Linux, Xbox, PlayStation, and mobile devices.

One of the key benefits of Unreal Engine is its ease of use and accessibility to developers of all skill levels. It provides a large and active community of developers and users, as well as extensive documentation and tutorials to help developers get started quickly. The scripts were written in Visual Studio 2022 with the c++ programming language.

Unreal Engine Marketplace provides a wide range of high-quality assets and resources to Unreal Engine users. It's got everything from a character, environment, or animation to sound effects, music and plugins. Developers are also able to make their own assets available on the marketplace, making it easier for them to generate income from their skills and work.

2.1.2 Microsoft Azure Kinect DK

The Microsoft Azure Kinect DK(Developer Kit) [2] is a sensor package designed for computer vision and speech models. To capture high quality data for computer vision and speech applications, it is the combination of depth sensors, a high definition RGB camera with multiple microphone inputs that can be used.

The Microsoft Azure DK Kinect provides developers with a wide range of features, which allow them to create cutting edge computer vision and speech applications. Some of the main features are as follows: Depth Sensor: The depth sensor enables developers to create 3D models of objects, detect and track movement, and enable gesture recognition. High definition RGB camera: High resolution video and pictures can be captured by the High-Definition RGB camera which is capable of recognizing objects as well as tracking them.

Microphone Array: The microphone array offers excellent speech recognition capabilities, allowing developers to develop applications that can recognize commands and react in accordance with them. Software development kit: In order to help developers create apps fast and easy, the Azure Kinect DK will come with a software development kit containing APIs, code samples or other documentation.

2.1.3 PCL Library C++

The Point Cloud Library (PCL) [3] is an open-source library for processing 2D and 3D point clouds. It was developed by a community of researchers and engineers from various universities and companies, and is widely used in computer vision, robotics, and 3D printing. A number of algorithms and tools for working with point cloud data are available in PCL, such as filtering, segmentation, registration or surface reconstruction. It supports a number of common file formats for point cloud data, such as PCD, POY, OBJ, STL, etc. The library is compiled in C++, and contains API for Python, Java, and MATLAB among other programming languages. A number of platforms, such as Windows, Linux or macOS, are supported by PCL.

2.1.4 Azure Kinect SDK

Azure Kinect SDK **Σφάλμα! Το αρχείο προέλευσης της αναφοράς δεν βρέθηκε.** is a collection of software tools and libraries that enables developers to create applications that use data from the Azure Kinect sensor. The Azure Kinect sensor is a high-end depth camera designed for computer vision and robotics applications.

The SDK includes APIs that enable you to access sensor data in depth, color, and infrared as well as perform body tracking or other computer vision tasks. In order to speed up the development process, it includes a set of sample applications and code snippets.

The Azure Kinect SDK supports C++, C# and Python programming languages that can be used on both Windows and Linux. The SDK is free to use, available on GitHub, making it possible for developers to contribute and edit their own code in order to suit their individual needs.

2.1.5 Python Open3D Library

The Open3D library[6] is an open-source 3D data processing library which was first released in 2018. It includes a set of powerful, easy to use tools for the generation, manipulation and simulation of 3D geometry and simulating point clouds. Open3D is created in C++ and offers Python bindings, so it can be used by both C++ and Python developers.

Some of the key features of Open3D include:

- 3D data types: a range of 3D data types, such as Pointcloud, TriangleMesh, VoxelGrid and Images are available from Open3D.
- A large range of 3D processing tools, such as a point cloud registration, mesh reconstruction, segmentation and more are available in Open3D.
- Visualization: Open3D provides a set of easy-to-use visualization tools that allow you to visualize your 3D data in 3D space.
- IO: Open3D supports importing and exporting various 3D file formats, including PLY, OBJ, STL, and OFF.
- Integration with deep learning: Open3D has integration with popular deep learning frameworks such as PyTorch and TensorFlow, making it a useful tool for tasks such as 3D object detection and segmentation.

In general, Open3D is a powerful and easy to use library which has become increasingly popular amongst 3D Data Processing Enthusiasts and Researchers.

2.1.6 OpenCV library

OpenCV (Open-Source Computer Vision)[7] is an open-source library that provides tools and functions for computer vision and image processing tasks. It was originally developed by Intel and later supported by Willow Garage and Itseez.

OpenCV is widely used in various fields such as robotics, augmented reality, facial recognition, object detection, and video analysis. The library supports multiple programming languages, including C++, Python, Java, and MATLAB, making it accessible to developers in different domains.

Here are some key features and functionalities provided by OpenCV:

- Image and video I/O: OpenCV can read and write images and videos from various file formats.
- Image processing: It offers a wide range of functions for manipulating images, including resizing, cropping, filtering, and transforming images.
- Feature detection and extraction: OpenCV provides algorithms to detect and extract features from images, such as corners, edges, and keypoints.

- Object detection and recognition: The library includes pre-trained models and methods for object detection and recognition tasks, such as Haar cascades, HOG (Histogram of Oriented Gradients), and deep learning-based approaches.
- Camera calibration: OpenCV allows calibration of cameras to correct for lens distortion and obtain accurate measurements from images.
- Machine learning support: OpenCV integrates with popular machine learning frameworks like TensorFlow and PyTorch, enabling the use of trained models for various computer vision tasks.
- Video analysis: It provides functions for video stabilization, motion tracking, and background subtraction.
- GUI and visualization: OpenCV offers graphical user interface (GUI) components to display and interact with images and videos.
- Parallel computing: OpenCV utilizes multi-core processors and hardware acceleration to optimize performance for computationally intensive tasks.

2.2 Challenges

2.2.1 Storage and Compression

The enormous amount of information required makes storage and compression of volumetric data extremely difficult. Raw volumetric data may be difficult to store and send, as well as being computationally expensive. Therefore, effective volumetric data compression techniques are being created to minimize storage needs and enable real-time streaming of collected material. These compression methods make an effort to maintain visual integrity while balancing compression ratios, guaranteeing that volumetric data may still be accessed and used without suffering significantly.

2.2.2 Real-Time Rendering

Volumetric content rendering in real time is a difficult undertaking because of its complexity and computing demands. To retain realism, volumetric data has to be rendered from a variety of angles with precise lighting and shading. To reach interactive frame rates for volumetric content, researchers are investigating strategies including level-of-detail representations,

hierarchical data structures, and GPU-based rendering algorithms. Advancements in real-time rendering are essential for providing interactive experiences, live performances, and dynamic virtual worlds with volumetric capture.

2.2.3 Data Acquisition and Processing

The effective acquisition and processing of massive volumes of data presents one of the main technological hurdles in volumetric capture. Massive data sets with many perspectives, depth information, and color data are produced by volumetric capture devices. High-resolution and high-fidelity volumetric material demands significant processing and storage power. To address these issues and improve accessibility of volumetric capture, researchers are currently investigating data compression methods, effective data formats, and distributed computing options.

2.2.4 Hardware and Sensor Limitations

The cost, scalability, and mobility of the hardware and sensor technologies utilized in volumetric capture systems might be drawbacks. To collect volumetric data, multi-camera systems, depth sensors, and specialized capture rigs are frequently employed. To lessen the complexity and expense of volumetric capture systems, researchers are looking into alternative hardware configurations such single-sensor depth estimation methods. Volumetric capture may become more affordable and extensively used as a result of improvements in consumer-grade depth sensors and hardware downsizing.

2.2.5 Depth and Occlusion Challenges

Volumetric capture faces continual difficulties with controlling occlusions and accurately recording depth information. Incomplete or incorrect representations may result from depth sensors' inability to handle obstructed regions and capture fine features. To increase depth estimate accuracy and effectively handle occlusion, researchers are looking into new algorithms and sensor fusion approaches. To solve these issues and improve the fidelity of volumetric capture, methods including multi-view stereo reconstruction, depth inpainting, and sensor fusion techniques are being investigated.

2.3 Scene Description

2.3.1 The Metahuman Character of myself

The first scene is the metahuman character of myself created with photos from different angles and with the help of Kentool Facebuilder[5] and the metahuman plugin. Details of the process followed to create the character will be discussed in the next chapter, chapter 3.



Figure 2 - Metahuman Character of myself (right angle)



Figure 1 - Metahuman Character of myself (front)

the character can be easily animated since it has a skeleton - rig and so you can control exactly the position, and the rotation of the movement of each point. For example to lift up the character's arm or open his mouth

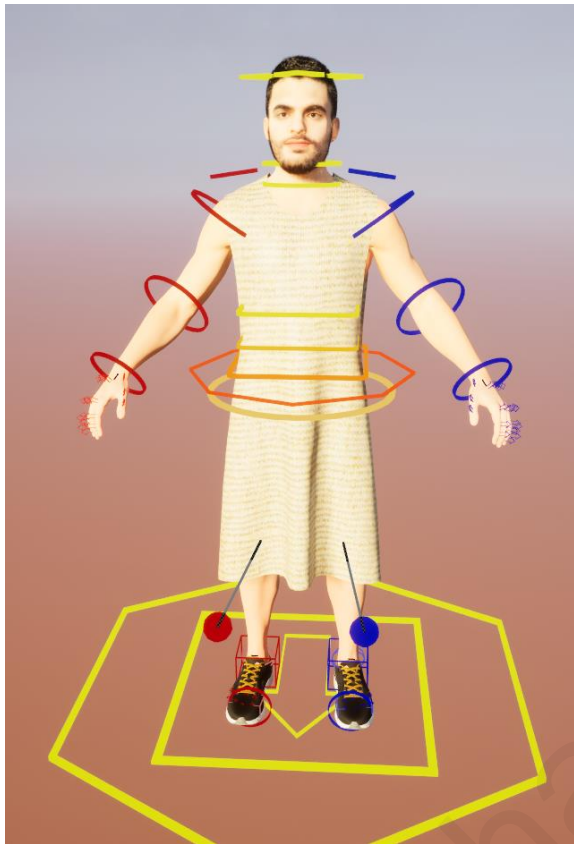


Figure 4 - Metahuman Character Control Rig – Up Arm

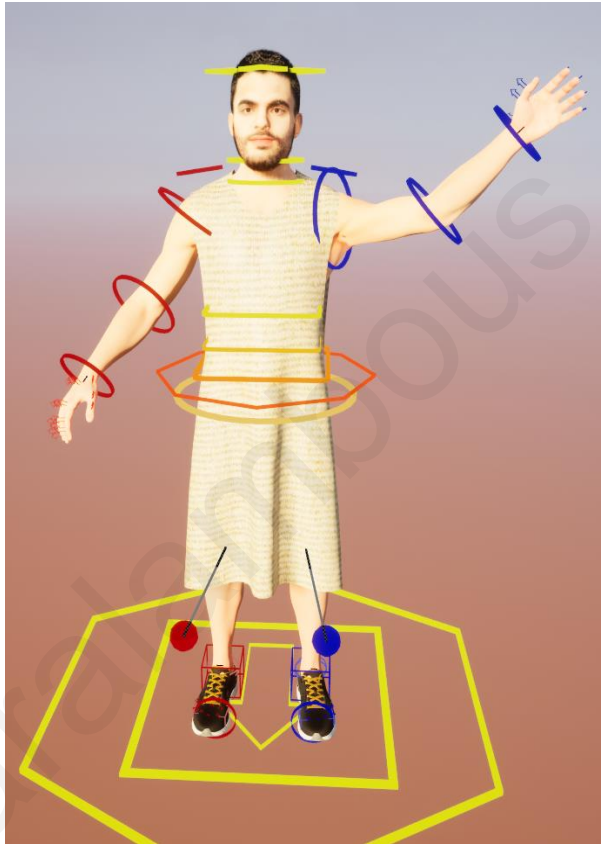


Figure 3 - Metahuman Character Control Rig – Up Arm

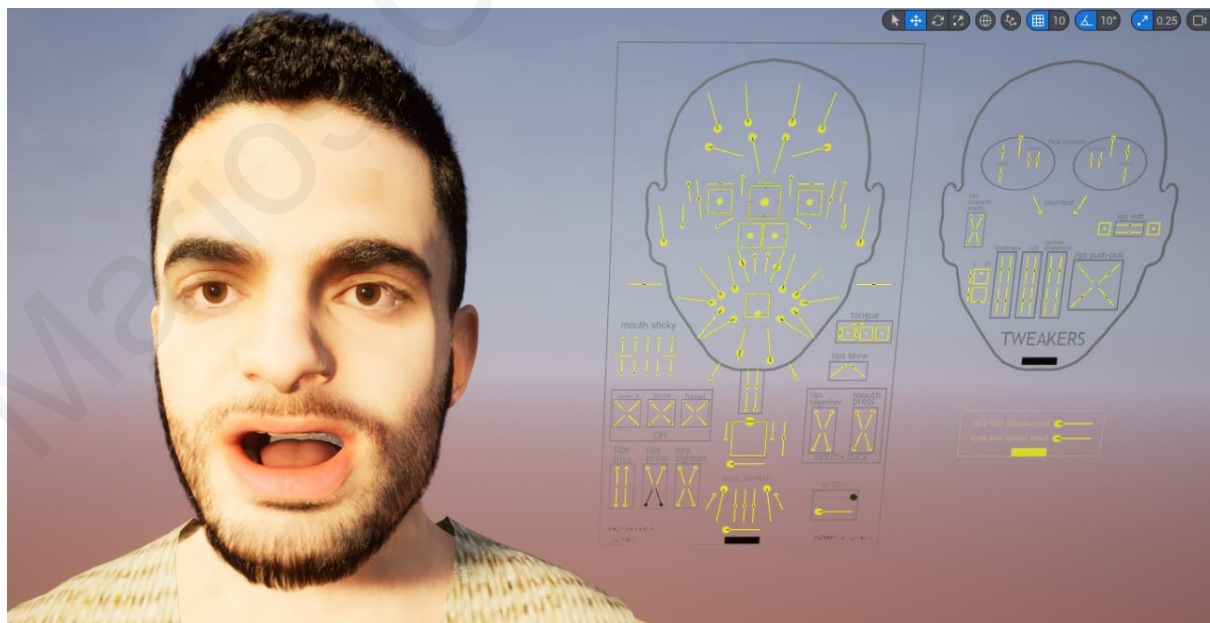


Figure 5 - Metahuman Character Control Rig – Face

2.3.2 Scene 2 – Realtime Volumetric Capture in Unreal Engine

In the second scene the only actor present is the Registration Manager where it is responsible to first find out how many Microsoft Azure Kinect depth sensors are connected to the computer and create the correct number of Azure Kinect Actor. It is also responsible for the following operations:

- Should Register Multiple Kinects
- Save Registration
- Register to World
- Record Point Cloud

And finally, close all depth sensors that were previously opened. The analysis and explanation of the operations will be done in chapter 4.



Figure 6 - Initialization of the three azure kinect



Figure 7 - Start Capturing

Chapter 3

3 Methodology

3.1 Process to create Metahuman Character of myself	19
3.2 General overview how to create volumetric character with multiple depth sensors	23
3.2 Setting up of Microsoft Azure Kinect depth sensors	24
3.3 Capturing the character – object	26
3.4 Merging point cloud - Calibration	30
3.5 Processing the data	32
3.6 Animating the character	35

3.1 Process to create Metahuman Character of myself

For the metahuman creation used KeenTool FaceBuilder for blender and the metahuman plugin offered by epic games for the unreal engine. Initially the first step take high-quality photos of the human face from multiple angles and it is very important that the lighting in the room is ambient to avoid reflections and shadows that will ruin the textures will use to create the metahuman.



Figure 8 - high-quality photo(front)



Figure 9 - high-quality photo(left angle)



Figure 10 - high-quality photo(right side)

Align Face

A couple of neural networks will find a face on the photo and set up some pins to match the position and the shape based on facial landmarks(mouth, ears ,nose, eyes, eyebrows).



Figure 11 - align face - left

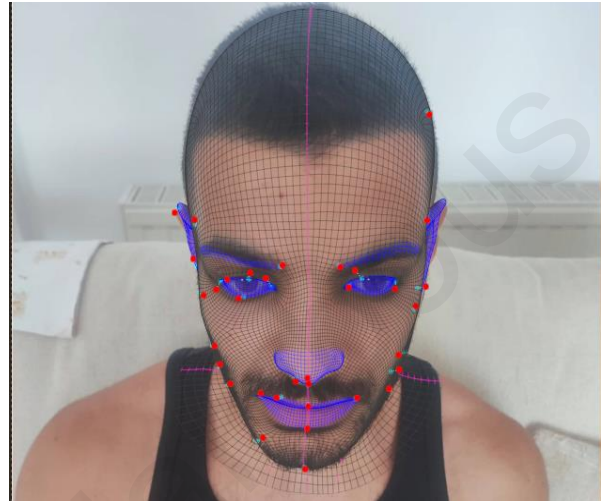


Figure 12 - align face - up

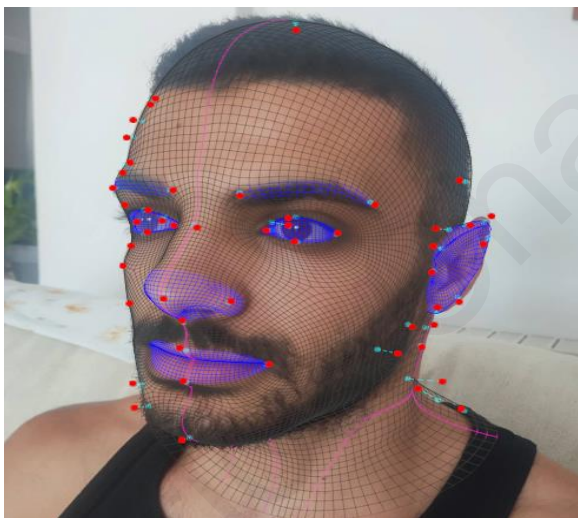


Figure 13 - align face - left

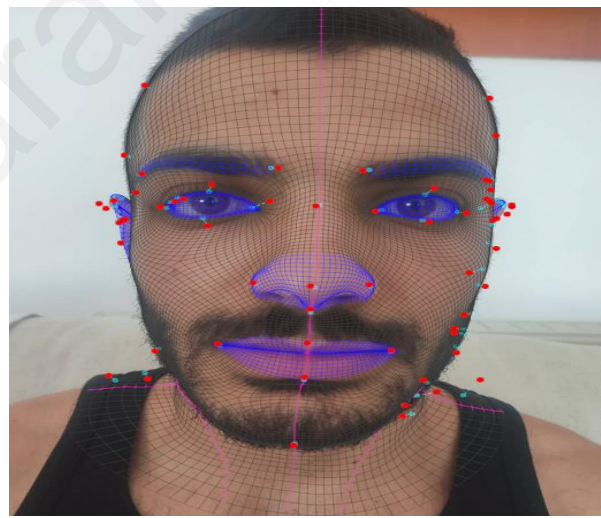


Figure 14 - align face - front

Once the head and face are pinned across the desired number of views, FaceBuilder can create the texture from the photos automatically. And final export the model as an FBX file which will load to Unreal Engine.



Figure 15 - Kentool Mesh



Figure 16 - Kentool mesh with texture



Figure 17 - Kentool texture

Mesh to MetaHuman: Aligning the model

At this stage the mesh must become a metahuman, that is, it must have the topology of the metahuman in order to be animated easily. Then frame the face to see the frontal view with all its details and launch auto-tracking, which takes a couple of moments and gives us the automatically detected facial landmarks.

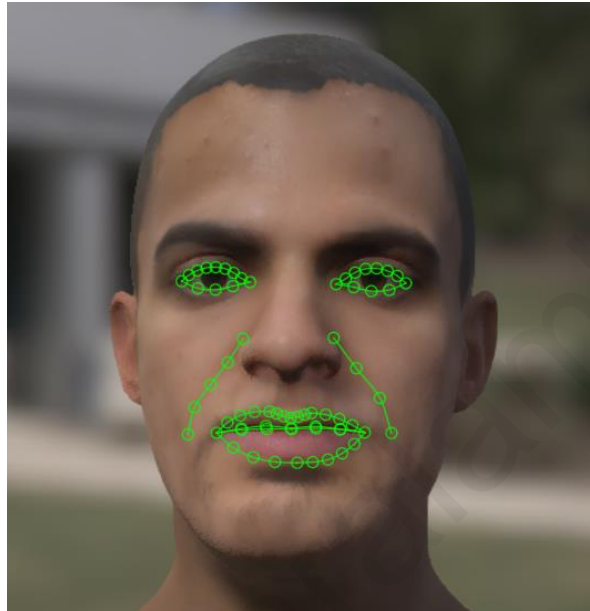


Figure 18 - Align Face to create Metahuman

MetaHuman Creator: set up the appearance of the character

In the online MetaHuman editor, choose the character and tune it to match the person from the photos as closely as possible by tweaking the skin colour, eyes, facial hair, clothes and so on. Using the online editor, you can reach a very good level of likeness, but there's a way to make it even better. To achieve this replace the generated MetaHuman texture with something more realistic and feature-rich.

Photoshop: Preparing the texture for MetaHuman

MetaHuman uses four textures for different facial expressions including the neutral one. Take them all to Photoshop and merge with the mh texture previously extracted from the photos by FaceBuilder. The mh texture map of FaceBuilder was created specifically to make this process as simple as possible. It matches the layout of the built-in MetaHuman texture perfectly, so all facial parts will align naturally, just need to set up the size of the texture properly while creating and exporting the texture. At this stage, need to accurately join high and low-frequency details

from both textures and match their colours. Also clean up the standard MetaHuman texture from the features that it has, but your real person hasn't.

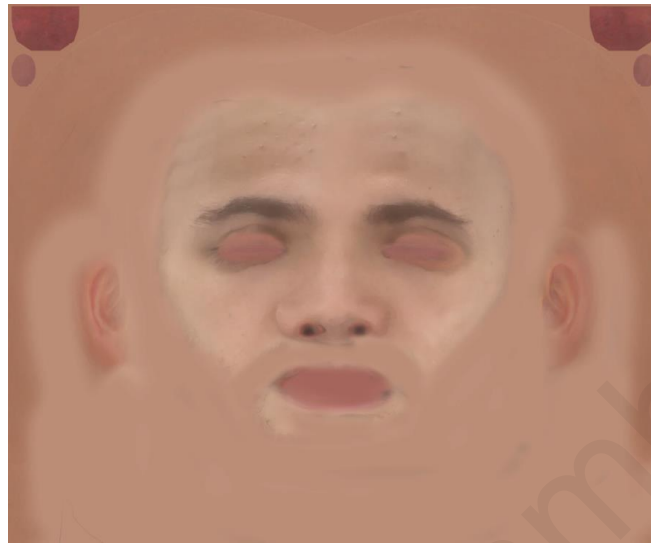


Figure 19 - Final texture

Unreal Engine: adding the new textures

The final step ,replace the textures of the model with the ones created earlier in Photoshop out of the MetaHuman ones and the ones extracted from the photos by FaceBuilder. Do it one by one for every expression.

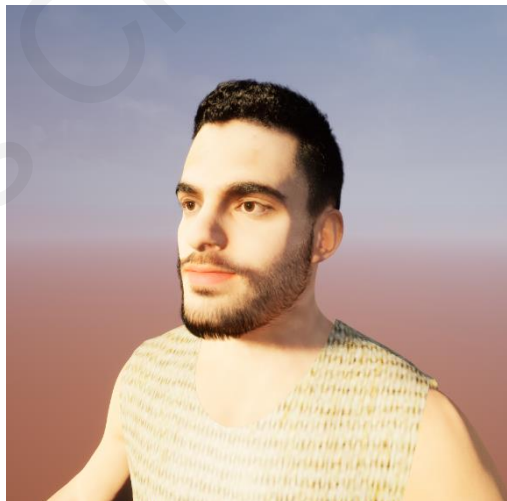


Figure 20 - Metahuman of myself

3.2 General Overview how to create volumetric character with multiple depth sensors in realtime

Creating a volumetric capture character with multiple depth sensors in realtime involves using several devices to capture the subject from different angles and perspectives. A general overview of the steps required to create a volumetric capture character with multiple depth sensors is given below:

1. Setting up of depth sensors: It shall be necessary to fix a number of depth sensors and recalibrate them so that they are able to capture the correct depth information. In this case, the sensor shall be placed at a precise distance from the subject and its settings adjusted to ensure best capture.
2. Capturing the character - object: The character-object shall be placed in a captured area, which by definition is an equatorial or polarised space to allow for simultaneous capture from multiple angles. The depth sensors capture both the color and depth information of the character - object, creating multiple point clouds of the character's shape and position.
3. Merging the point clouds: The captured data from each depth sensor is then merged together using a calibration technique. This involves aligning multiple point clouds to a single, unified 3D model of the object.
4. Processing the data: The point clouds captured by the sensors need to be processed to create a smooth and accurate surface representation of the character. This typically involves performing various steps such as filtering, meshing, and surface reconstruction.
5. Animating the character: Once the volumetric representation of the character is created, it can be animated in real-time. This involves mapping the motion of the sensors to the character's surface, allowing the character to move in real-time as the sensors capture its movement.

3.3 Setting up of Microsoft Azure kinects depth sensors

Initially very important was the positioning of the cameras. Depending on the depth mode of the camera that the user wants to choose (NFOV or WFOV) the cameras must be placed at a different distance for better capture of space and character. For NFOV depth mode the cameras

must be placed above the ground at least 0.7 meters and no more than 1.5 meters and the person at a distance of more than 1.5 meters away from the camera for a complete capture of his body.

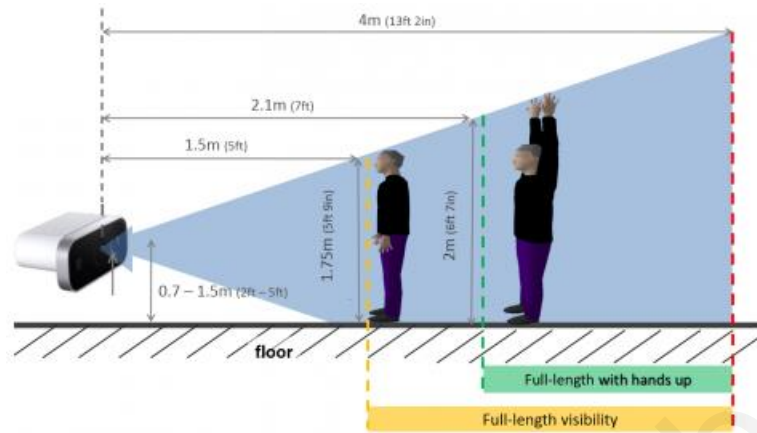


Figure 21 - NFOV distances

For WFOV depth mode the cameras must be placed above the ground at least 0.7 meters and no more than 1.5 meters and the person at a distance of more than 0.8 meters away from the camera for a complete capture of his body.

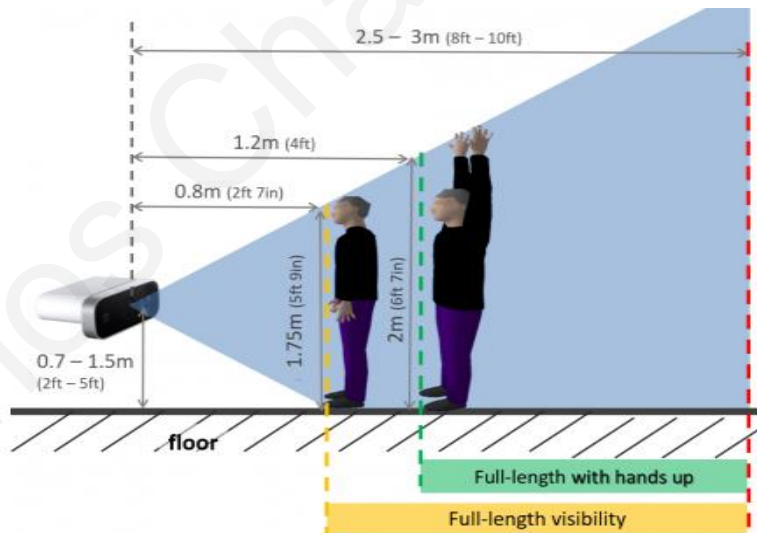


Figure 22 - WFOV distances

Apart from the height and distance that the user must have from the camera, there is also the angle and distance that each camera must have between each other. For NFOV depth mode ideally, depth sensors should be placed at apexes of equilateral triangle with sides equal to

about 5 meters and for WFOV depth sensors should be placed at apexes of equilateral triangle with sides equal to about 3 meters.

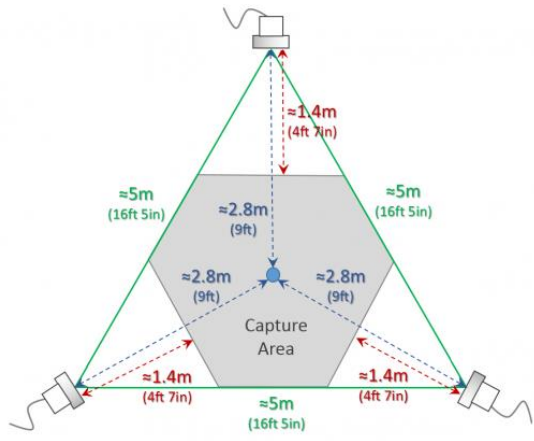


Figure 24 - Ideal configuration for triple azure kinects sensors(NFOV)

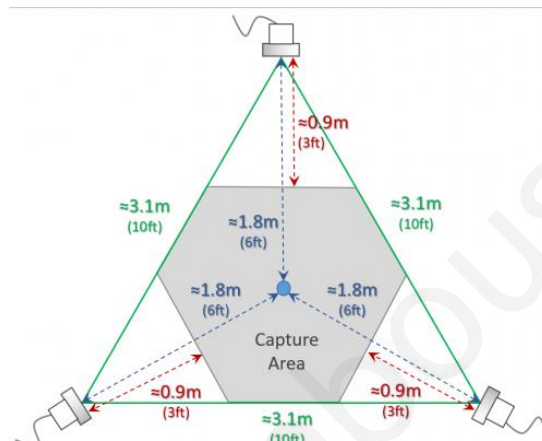


Figure 23 -Ideal configuration for triple azure kinects sensors(WFOV)

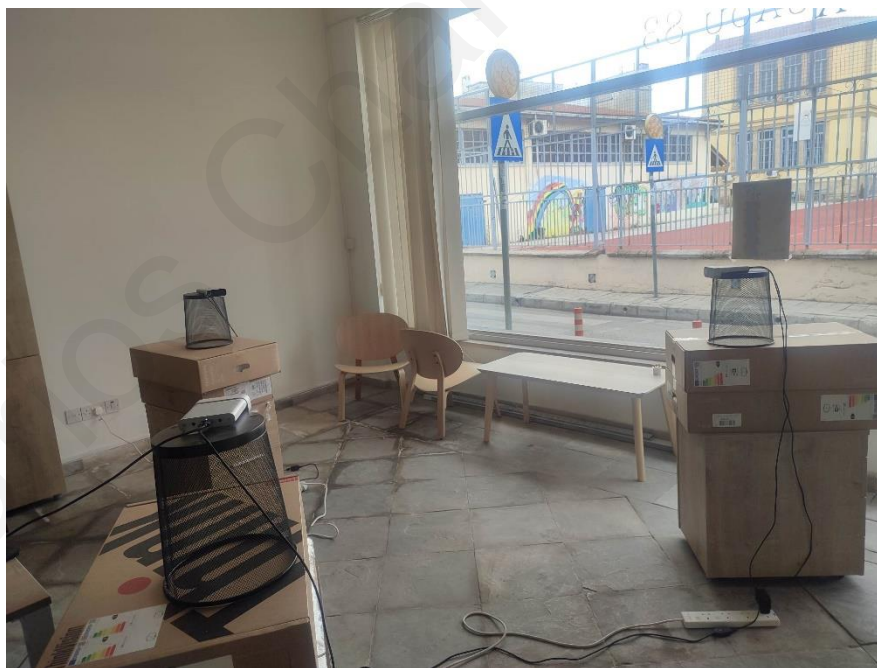


Figure 25 - Kinects Setup

In the picture above you can see the set up of the depth sensors.

3.4 Capture the character - object

After the correct placement of the cameras, the process of character capture and creation of point clouds for each individual camera follows. The steps for this process are:

1. Initialize the Azure Kinect device and configure its settings. This includes specifying the color format and resolution, depth mode, and camera FPS.
2. Retrieve the device's calibration data, which provides information about the intrinsic and extrinsic parameters of the color and depth cameras.
3. Create a transformation object that can convert depth images to point clouds using the calibration data.
4. Start the device and retrieve a capture, which contains the latest color and depth images.
5. Extract the depth and color images from the capture.
6. Use the transformation object to convert the depth image to a point cloud image.
7. Access the point cloud data to extract the 3D coordinates of the points in the cloud.
8. Perform any additional processing or analysis on the point cloud data as needed.(median filter at point cloud image for remove the noise – outliers points)
9. To Render the point cloud into the scene in unreal engine the easiest way is use Niagara Point Cloud System. The position of each point is the depth image and the color of each point is the color image. Configure the renderer, set the "Point Size" and "Minimum Pixel Size" parameters to control the size of the rendered points.
10. Release all resources when finished, including the device, calibration data, transformation object, and images.

Pseudocode

```
// Initialize the Kinect sensor and transformation
k4a_device_t device = NULL;
if (k4a_device_open(0, &device) != K4A_RESULT_SUCCEEDED) {
    printf("Failed to open device\n");
    return;
}
k4a_device_configuration_t config = K4A_DEVICE_CONFIG_INIT_DISABLE_ALL;
config.color_format = K4A_IMAGE_FORMAT_COLOR_BGRA32;
```

```

config.color_resolution = K4A_COLOR_RESOLUTION_720P;
config.depth_mode = K4A_DEPTH_MODE_NFOV_UNBINNED;
if (k4a_device_start_cameras(device, &config) != K4A_RESULT_SUCCEEDED) {
    printf("Failed to start cameras\n");
    k4a_device_close(device);
    return;
}
k4a_calibration_t calibration;
if (k4a_device_get_calibration(device, config.depth_mode, config.color_resolution,
&calibration) != K4A_RESULT_SUCCEEDED) {
    printf("Failed to get calibration\n");
    k4a_device_stop_cameras(device);
    k4a_device_close(device);
    return;
}
k4a_transformation_t transformation = k4a_transformation_create(&calibration);
while (true) {
    // Capture a color and depth frame from the Kinect sensor
    k4a_capture_t capture = NULL;
    if (k4a_device_get_capture(device, &capture, K4A_WAIT_INFINITE) !=
K4A_RESULT_SUCCEEDED) {
        printf("Failed to get capture\n");
        continue;
    }
    k4a_image_t color_image = k4a_capture_get_color_image(capture);
    k4a_image_t depth_image = k4a_capture_get_depth_image(capture);
    // Generate the point cloud
    k4a_image_t undistorted_depth_image = NULL;
    if (k4a_image_create(K4A_IMAGE_FORMAT_DEPTH16, width, height, width *
(int)sizeof(uint16_t), &undistorted_depth_image) != K4A_RESULT_SUCCEEDED) {
        printf("Failed to create undistorted depth image\n");
        k4a_device_stop_cameras(device);
        k4a_device_close(device);

```

```

    return;
}
k4a_image_t transformed_color_image = NULL;
if (k4a_image_create(K4A_IMAGE_FORMAT_COLOR_BGRA32, width, height, width
* (int)sizeof(uint32_t), &transformed_color_image) != K4A_RESULT_SUCCEEDED) {
    printf("Failed to create transformed color image\n");
    k4a_device_stop_cameras(device);
    k4a_device_close(device);
    return;
}
k4a_transformation_depth_image_to_color_camera(transformation, depth_image,
undistorted_depth_image);
k4a_transformation_color_image_to_depth_camera(transformation, depth_image,
color_image, transformed_color_image);

// Get the color data
uint8_t* color_data = k4a_image_get_buffer(transformed_color_image);
int color_stride = k4a_image_get_stride_bytes(transformed_color_image);
int color_width = k4a_image_get_width_pixels(transformed_color_image);
int color_height = k4a_image_get_height_pixels(transformed_color_image);
// Allocate memory for the point clouds
k4a_float3_t* point_cloud = new k4a_float3_t[width * height];
// Generate the point cloud
for (int i = 0; i < width * height; i++) {
    int x = i % width;
    int y = i / width;
    k4a_float2_t undistorted_point = k4a_float2_t{ (float)x, (float)y };
    uint16_t depth_value = *reinterpret_cast<const
uint16_t*>(k4a_image_get_buffer(undistorted_depth_image)) +
(k4a_image_get_stride_bytes(undistorted_depth_image) * y) / sizeof(uint16_t) + x;
    k4a_float3_t point = k4a_calibration_2d_to_3d(&calibration, &undistorted_point,
depth_value, K4A_CALIBRATION_TYPE_NFOV_UNBINNED);
    point_cloud[i] = point;
}

```



```

// Release resources
k4a_image_release(color_image);
k4a_image_release(depth_image);
k4a_image_release(undistorted_depth_image);
k4a_image_release(transformed_color_image);
delete[] point_cloud;
k4a_capture_release(capture);

```



Figure 26 - Point Cloud rendering in unreal engine

3.5 Merging point cloud - Calibration

Up until this point, you create a separate point cloud for each Kinect depth sensor. What needs to be done is merge into one point cloud.

Methods 1 chessboard pattern:

To calibrate multiple Azure Kinect cameras using a chessboard pattern, you can follow these general steps:

1. Capture images of a chessboard pattern(see Fig.23) using each of the Azure Kinect cameras. It's best to capture the images in a well-lit environment to ensure that the chessboard pattern is clearly visible in the images.
2. Use a calibration toolbox like OpenCV to extract the corners of the chessboard pattern in each image. Will

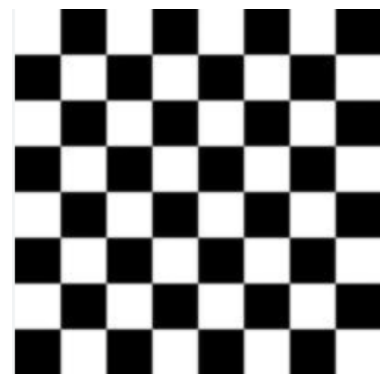


Figure 27 - Chessboard pattern

need to specify the dimensions of the chessboard squares in millimetres.

3. Use the extracted chessboard corners to calibrate each of the Azure Kinect cameras individually. This can be done using a standard calibration routine, such as the `cv2.calibrateCamera()` function in OpenCV.
4. Once have the intrinsic camera parameters (like focal length and principal point) for each camera, can use a stereo calibration routine to compute the extrinsic parameters (like rotation and translation) that describe the relative pose of the cameras with respect to each other. This can be done using the `cv2.stereoCalibrate()` function in.
5. Finally, can use the computed extrinsic parameters to rectify the images and create a depth map. This can be done using the `cv2.stereoRectify()` function in OpenCV.

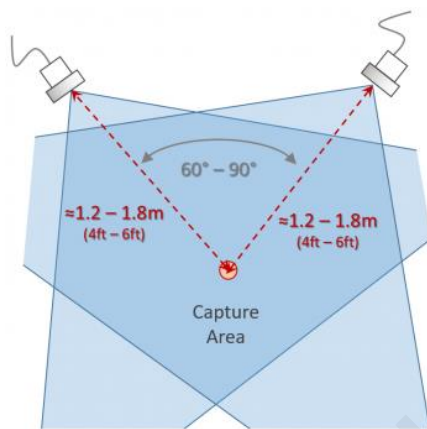


Figure 28 - Configuration for chessboard pattern

The problem with this method was when the depth sensors had to be placed next to each other, because they had to be able to see the chessboard. This way all the information of the object or character on one side was lost.

Methods 2 lattice pattern:

Instead of using a checkerboard pattern in an infrared image (IR) for calibration, created a lattice with evenly spaced rectangular holes that can be detected directly in a depth image. This lattice consists of 25 evenly spaced rectangular holes (4cm x 4cm each) (see Fig. 23).

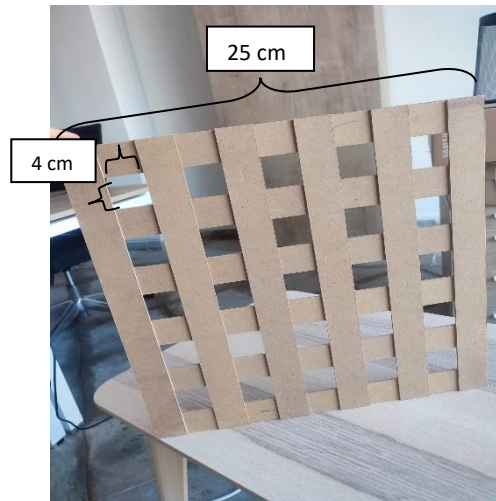


Figure 29 - lattice

First, look for gaps in the depth image that correspond to the lattice's geometry. Then, cluster these gap segments to determine the physical lattice's region, filtering out clusters that do not meet certain thresholds. Use proximity to gap segments to identify 3D points that could belong to the lattice, and filter out noise using RANSAC to fit a plane and classify points as inliers or outliers. Only after this do determine the precise locations of the holes in 3D. Identify the holes by iterating over points and considering those that are not inliers and are circumscribed by inliers. Merge segments belonging to the same hole using the union find structure, and estimate the centers of each hole by averaging the inliers in the neighborhood of the hole segments.

In order to determine the orientation of the lattice, a heuristic that can handle even noisy hole centers, which will be called grid nodes in the following. In order to do so, define the following set of vectors: $V = \{n - m \mid l < \text{dist}(m, n) < h, m \in H, n \in H\}$

where $l = d - \delta, h = d + \delta, d = \text{lattice spacing}$ and δ is a tolerance ($d = 8\text{cm}$ and $\delta = 2\text{cm}$),.

This set is clustered by the angle the vectors subtend with the x-axis. The two largest clusters represent the prevalent directions; the median of each is considered the direction of the x- and y-axis of the lattice. Now have the two very stable directions of the calibration lattice in space, but don't know which one is which axis and their signs. To resolve this ambiguity, consider the hands holding the lattice to estimate the orientation and align the x-axis such that it points towards the hands. For the third z-axis use the normal of the plane determined by RANSAC earlier, and align it towards the camera. Finally, register the depth cameras using the SVD-based transformation estimation of the Point Cloud Library[15].



Figure 30 - Merging Point cloud

3.6 Processing the data

The merged point cloud before it becomes a mesh has to be filtered to clean up the noise it has (to remove outliers). This was done with the median filter. A median filter is a type of digital signal processing filter commonly used in image processing and digital audio signal processing. The purpose of the median filter is to reduce noise in the signal by replacing each pixel value with the median value of adjacent pixels within a given window.

A median filter works by sliding a window of a specified size over the input signal and computing the median of the pixel values within the window for each window position. This median value is used to replace the original pixel value in the center of the window.

Compared to other filter types, median filters are a type of noise that can appear in an image or audio signal when individual pixels or samples are randomly aliased to very high or very low values. It is particularly effective in reducing “salt and paper” noise, which is A median filter is a nonlinear filter. That is, the output depends not only on the input signal, but also on the properties of the filter itself. Nonlinear filters can produce artifacts and other undesirable effects, so it is important to choose an appropriate filter size and other parameters when using median filters. So when applied over the textures (depth and color) created by the sensors, it cleared the merged point cloud.

Median filter Pseudocode

1. allocate outputPixelValue[image width][image height]
2. allocate window>window width × window height
3. edgex := (window width / 2) rounded down
4. edgey := (window height / 2) rounded down
for x from edgex to image width - edgex do
for y from edgey to image height - edgey do
i = 0
for fx from 0 to window width do
for fy from 0 to window height do
window[i] := inputPixelValue[x + fx - edgex][y + fy - edgey]
i := i + 1
sort entries in window[]
outputPixelValue[x][y] := window>window width * window height / 2]

At this point the mesh creation process can start.(Surface reconstruction). In computer graphics and computer vision, the process of surface reconstruction entails building a 3D surface model of an object or scene from a collection of 2D or 3D data points. This method is often employed in several fields, including robots, virtual reality, and medical imaging.

Surface reconstruction may be accomplished using a variety of strategies, including point-based techniques, implicit surface fitting, and parametric methods. A set of parameters, such as a collection of curves or surfaces, are used in parametric techniques to define the surface by fitting them to the input data points. In order to create a surface, implicit surface fitting entails fitting a mathematical function to the input data points.

This process required powerful hardware and therefore was done in non-real time. The positions of the points and colors (X Y Z R G B) (See Fig. 25) were exported in each frame and with the help of the open3d library a script was created in the python programming language where the steps of the process are as follows:

```
frame 0
4.66388 -26.5144 58.4 54 54 54
4.41244 -26.3679 58.4 45 44 45
4.54441 -26.4159 58.5 42 42 42
4.56 41 -26.56 59.5 43 43 44|
....
frame 1
21.3604 -23.5994 126.1 22 18 19
21.05 -23.5379 125.8 22 18 19
20.9394 -23.7009 126.7 26 22 23
25.1391 -22.6087 121.6 19 13 14
25.0486 -22.7511 122.4 20 13 16
....
```

Figure 31 - Record file

1. Load and prepare the data: takes the correct values for the positions of the points and colors (first three values for positions and the remaining three values for colors) and calculates the average distance between the points and their neighbours. This is done in order to estimate the normals so that it can create the final mesh.
2. Choose a meshing strategy: the first strategy is the Ball-Pivoting Algorithm and the second strategy is Poisson reconstruction.
 - Strategy 1 Ball-Pivoting Algorithm: The Ball-Pivoting Algorithm's (BPA) goal is to construct a mesh from a point cloud by simulating the usage of a virtual ball. Assume initially that the offered point cloud is made up of points that were taken as samples from an object's surface. For the rebuilt mesh to be explicit, points must rigorously represent a surface (and be noise-free). Using this assumption, imagine rolling a tiny ball across the point cloud "surface". Depending on the size of the mesh, this little ball should be a little bigger than the typical distance between points. A ball will get captured and land on three points that will eventually form the seed triangle if you drop it onto the surface of some points. The ball moves from that point along the edge of the triangle created by the two locations. The ball then settles in a new location: one new triangle is added to the mesh and a new triangle is created from two of the previous vertices. New triangles are formed and added to the mesh

as we keep rolling and pivoting the ball. The mesh is completely constructed when the ball stops moving.

- Strategy 2 Poisson reconstruction: The Poisson Reconstruction involves a little more math and technology. Its method, called an implicit meshing method, can be thought of as an attempt to "envelop" the data in a smooth fabric. Without getting into much detail, we attempt to create a watertight surface from the original point set by building a brand-new point set that represents an isosurface connected to the normals. There are several parameters available that affect the result of the meshing: Firstly Which depth? The reconstruction makes use of a tree-depth. The mesh has greater detail the higher it is (the default is 8). In a mesh that is created from noisy data, outlier vertices are still there but are not recognized as such by the algorithm. As a result, a low number (maybe between 5 and 7) smoothes the image, but you lose detail. The number of vertices in the created mesh increases as the depth-value increases. Which scale? It describes the ratio between the diameter of the cube used for reconstruction and the diameter of the samples' bounding cube. Very abstract, the default parameter usually works well (1.1). And finally which fit? the linear_fit parameter if set to true, let the reconstructor use linear interpolation to estimate the positions of iso-vertices.

3. Export and visualize: export both the BPA and Poisson's reconstructions as .ply files.

Results of Strategy 1



Figure 33 - Ball Pivoting mesh – frame 1



Figure 32 - Ball Bivoting Mesh - frame 2

Results of Strategy 2



Figure 35 - Poisson Reconstruction - frame 1



Figure 34- Poisson Reconstruction - frame 2

The results with the Poisson reconstruction method are better since the Ball Pivoting method lacks vertices and triangles, so gaps are created and the mesh surface is not good. In contrast with the Poisson reconstruction, we get an improved mesh surface without many artifacts and noise.

3.7 Animating the Character



Figure 37 - frame 1 animation



Figure 36 - frame 2 animation



Figure 39 - frame 3 animation



Figure 38 - frame 4 animation

The last step is animating the mesh. The construction of the mesh in each frame as discussed in the previous chapter (see 3.6) gives the character motion(see fig. 32 – 35).

Chapter 4

4 Operations

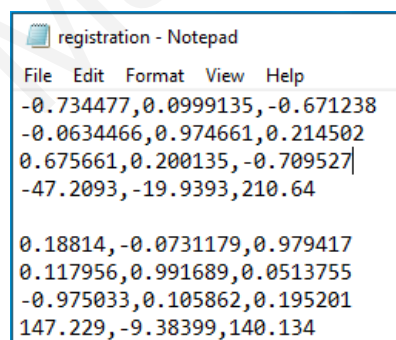
4.1 Should Register Multiple Kinects	42
4.2 Save Registration	42
4.3 Register to World	43
4.4 Record Point Cloud	43

4.1 Should Register Multiple Kinects

The first function is the Multiple Kinects Register which is responsible for calibrating the cameras and merging the three point clouds into one. The process of explaining the calibration was explained in chapter 3.4. First, cache parallel all the points from the point clouds and finds the grids for all three depth sensors. When detect the grids, match hole points of all grids. So, get results for registrate all the point clouds into the first point cloud.

4.2 Save Registration

the SaveRegistration function is responsible for saving to a Txt file the transformation matrixes needed to merge the point clouds. The procedure followed is simple, first it checks how many depth sensors are connected to the computer, and for each depth sensor, it saves its CurrentRegistration in the file.



```
registration - Notepad
File Edit Format View Help
-0.734477,0.0999135,-0.671238
-0.0634466,0.974661,0.214502
0.675661,0.200135,-0.709527
-47.2093,-19.9393,210.64

0.18814,-0.0731179,0.979417
0.117956,0.991689,0.0513755
-0.975033,0.105862,0.195201
147.229,-9.38399,140.134
```

Figure 40 - Registration file

The structure of each transformation is the first three lines are the Vector for the three axes (X, Y, Z) and the fourth line is the Vector for the translation.

4.3 Register to World

The RegisterToWorld function takes the registration file (see fig.30) saved in the previous function (SaveRegistration) and gives each Kinect the transformation it needs to automatically perform the Calibration without needing the lattice (see fig.25 chapter 3.5). The procedure followed is for each line in the file it splits the values in the vectors for the three axes (X, Y, Z) and for the translation in each ‘,’ it finds for each Kinect. Thus each Kinect becomes the correct transform to do the calibration automatically.

4.4 Record Point Cloud

The last function is RecordPointCloudToFile. This function stores in each frame the positions of each point on all three axes (X, Y, Z) and the color of each point (R, G, B). This is done for all point clouds created. When the user presses the RecordPointCloudToFile button the process starts and initially, the txt File(Record.txt) is created. Then, at the same time, it saves for each point cloud in each frame the positions of each point on all three axes(X, Y, Z) and the color that each point has(R, G, B). It stops the process when the user disables the RecordPointCloudToFile button.

```
frame 0
4.66388 -26.5144 58.4 54 54 54
4.41244 -26.3679 58.4 45 44 45
4.54441 -26.4159 58.5 42 42 42
4.56 41 -26.56 59.5 43 43 44|
....
frame 1
21.3604 -23.5994 126.1 22 18 19
21.05 -23.5379 125.8 22 18 19
20.9394 -23.7009 126.7 26 22 23
25.1391 -22.6087 121.6 19 13 14
25.0486 -22.7511 122.4 20 13 16
....
```

the first three columns are the position of each point and
the next three columns are the color of each point.

Figure 41 - Record File

Chapter 5

5 Manual

5.1 Scene 1 – The Metahuman Character of myself	44
5.2 Scene 2 – Realtime Volumetric Capture in UE	45

5.1 Scene 1 – The Metahuman Character of myself

Opening the first scene is the metahuman character and a cinemachine camera that tracks him. By pressing the start button, the animation of the character starts. Easily the user can create his own animation by opening the Animation Sequence inside the scene and choosing which bone he wants to rotate and move in whichever frame he wants.

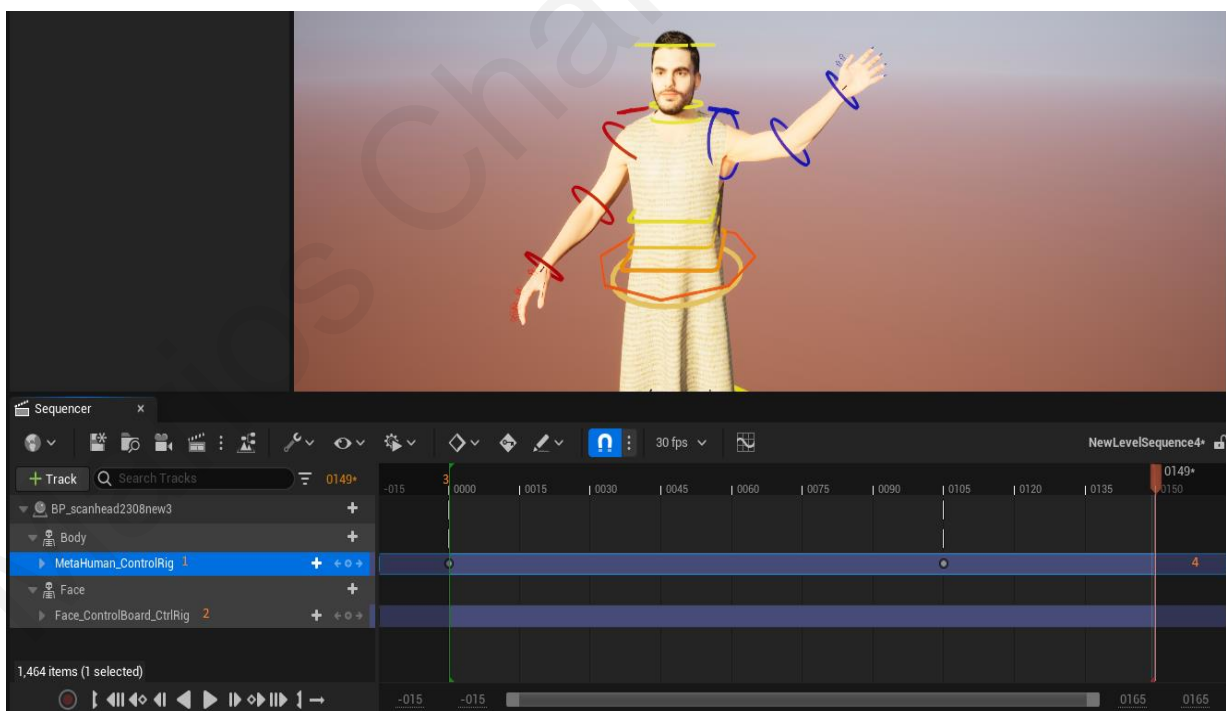


Figure 42 - Animation Sequence

in the picture above (see Fig.38) the ControlRig of metahuman is divided into face and body. First you choose which one you want to move and then you choose which bone you want from the two categories (face and body). Then you choose in which frame you want to start or end the movement of the bone you initially selected. This creates the animation and gives "life" to the character.

5.2 Scene 2 – Realtime Volumetric Capture in UE

Opening the second scene, only the registrationManager Actor containing the four functions is placed (see chapter 4) and pressing the Start button ,registrationManager Actor is responsible for finding how many Azure Kinect Depth sensors are connected on the computer and creating AzureKinect Actors to start capturing the space in a point cloud. Then he can choose the function he wants to perform by pressing the appropriate button. If he wants to merge the point clouds into one, then he chooses Should Register to Multiple Kinects(Fig.39 number 1) and with the help of the lattice pattern he does the calibration. Second function, if you choose Save Registration (Fig.39 number 2), it will save in a txt file the calibration matrix that the depth sensors have at that moment. Third operation, if the user selects Register to World (Fig.39 number 3), then it takes the matrices saved in the second mode and transforms the Kinects for automatic calibration. And the last function that the user can choose is the Record Point Cloud (Fig.39 number 4), where by pressing the button it starts simultaneously and saves in a file the positions and colors of all the points of the Point cloud and stops when the button is disabled. See chapter 4 for details on what each function does.

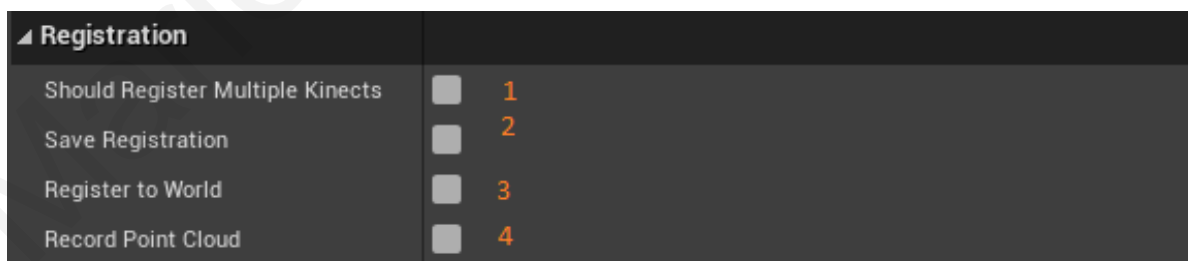


Figure 43 - Operations

Chapter 6

6 Conclusions – Results and Future Work

6.1 Conclusions – Results	46
6.1.1 Summary of the thesis	46
6.1.2 Discussion of the results	46
6.1.2.1 Chessboard vs Lattice	47
6.1.2.2 Calibration Mean Error for the positions of the depth sensors	49
6.1.2.3 Calibration Mean Error for lighting conditions.	53
6.1.2.4 Time needed for Calibration - lighting conditions.	54
6.1.2.5 Time and Calibration Mean Error – Change height	54
6.1.2.6 NFOV vs WFOV	55
6.1.2.7 Ball-Pivoting vs Poisson Algorithm	58
6.2 Future Work	61

6.1 Conclusions – Results

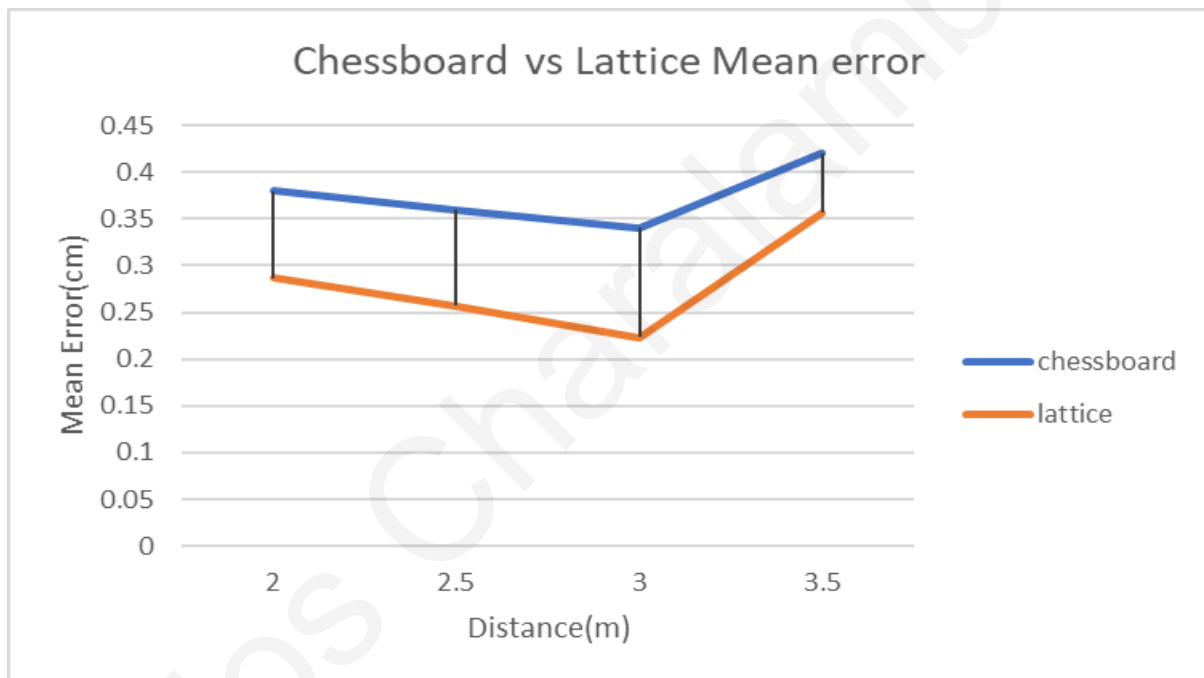
6.1.1 Summary of the thesis

In this thesis, the process of creating one's metahuman character with photos from different angles and having the topology of the metahuman avatar to be easily animated was first presented. In addition, it analyzed in detail the process of creating a volumetric capture system with multiple Azure Kinect depth sensors in real-time in the Unreal engine game platform. The process was initially to get the cameras set up in the correct position. Then, create each camera's point cloud and merge them into one. Then followed by processing the data where it was filtering - cleaning the point cloud and surface reconstruction where this process was done in non-real-time. And it was finally animating the character.

6.1.2 Discussion of the results

6.1.2.1 Chessboard vs Lattice

Chapter 3.5 presented the two methods used to merge the point clouds into one, to get the correct calibration on the cameras. The first method calibrates the cameras using the chessboard pattern and the infrared image (IR) as opposed to the second method which uses the depth image directly. The problem encountered with the first method was that the depth sensors had to be placed on the same side (side by side) to see the chessboard. This resulted in losing a lot of information about the object - character to be captured. On the other hand in the second method, the cameras can be placed anywhere.

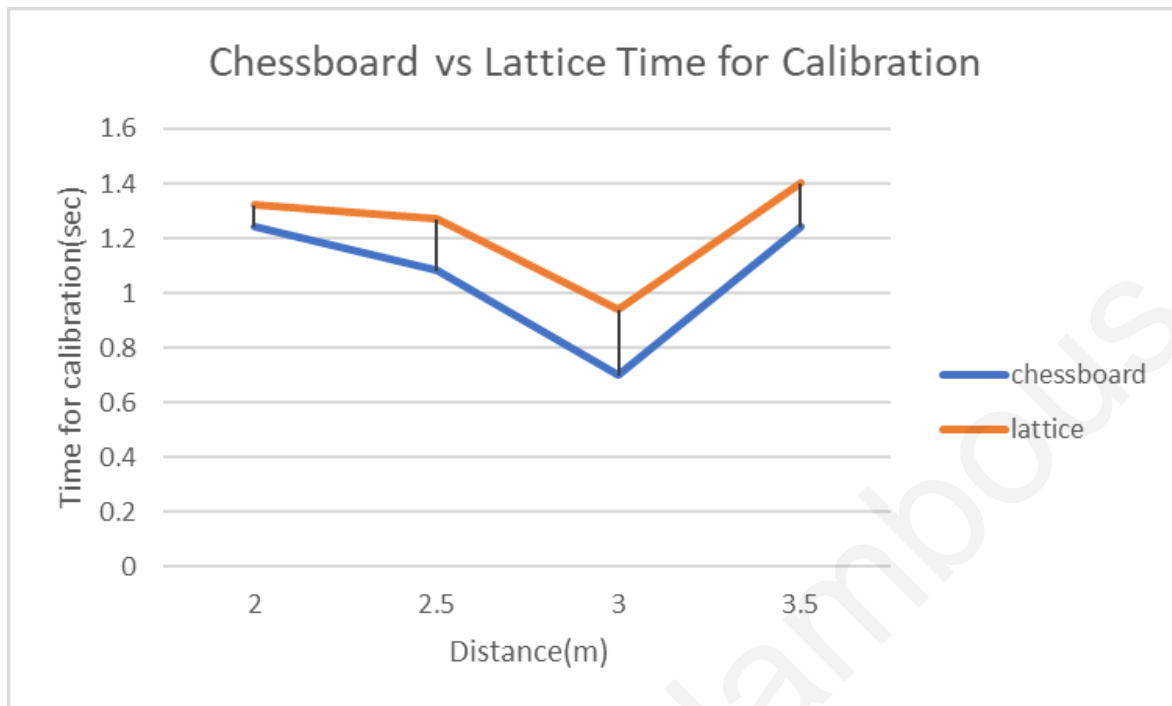


Graph 1 - Chessboard vs Lattice pattern (Mean Error)

The above graph compares the two methods in terms of mean error. The mean error is calculated as follows:

- Finds the distance of each point(sV to the corresponding point of the other point cloud(tV).
- sums these distances ($\text{sumError} += \text{Distance}(sV, tV)$).
- divides them by the size ($\text{meanError} = \text{sumError} / \text{size}$).

In both methods, the cameras were placed side by side at the same height(1 m) ,at distances of 2, 2.5, 3, and 3.5 meters and the mean error at these four distances was calculated. To determine the errors, the mean error was calculated six times for each distance and the average of the six values was plotted. As can be seen in both cases the results are almost the same with a small mean error. The best distance in both methods is 3 meters distance between the cameras with a mean error of just over 2 mm and the worst is 3.5 meters with a mean error of 0.35 mm.

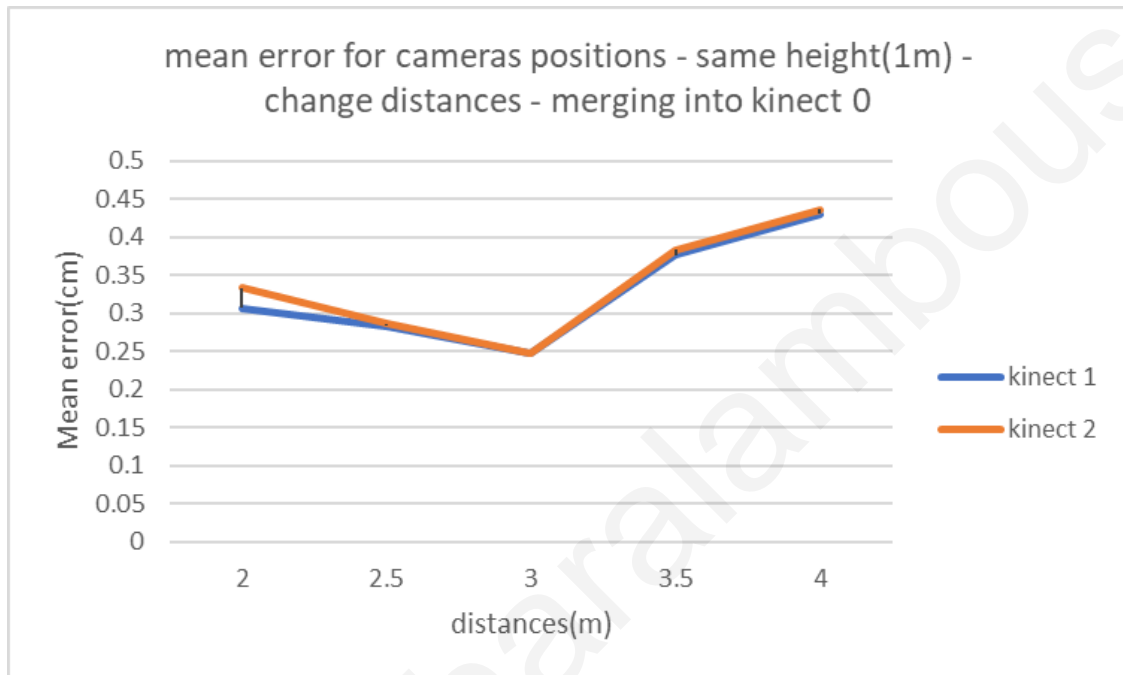


Graph 2 - Chessboard vs Lattice (Time needed for calibration)

In the above graph, there is another comparison of the two methods but this time it is done in terms of the time it took to calibrate the sensors. The time is calculated as soon as the calibration starts, i.e. the timer starts when the cameras see the chessboard and the lattice pattern respectively, and stops when the point clouds are aligned. Compared to the mean error, the time it took to do the calibration is much better in the second method (using depth image - lattice pattern) since at each distance calculated (2, 2.5, 3, 3.5 meters), it is about 0.35 sec faster. Again, in both methods the best distance is 3 meters with a time a little over 1.4sec and about 1sec respectively and the worst distance is 3.5 meters with times approximately 1.7sec and 1.4sec.

6.1.2.2 Calibration Mean Error for the positions of the depth sensors

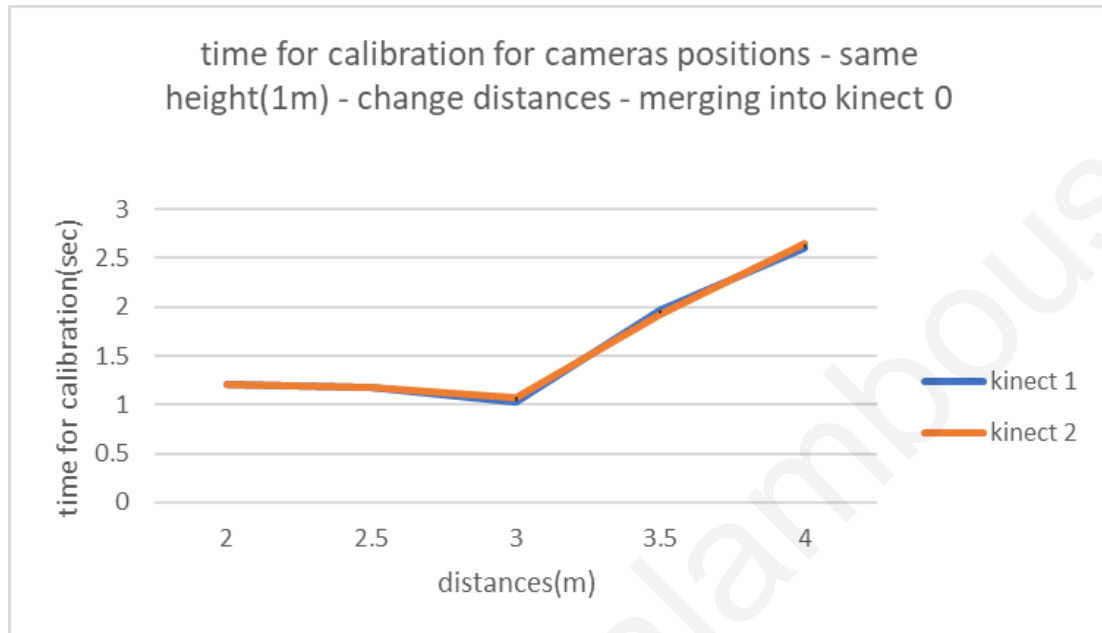
In the graph below (see Graph 3), the three cameras were placed in a triangle shape (see Fig. 19) with a constant height of 1 meter, changing the distance between them, the calibration mean error was calculated as explained previously (see chapter 6.1.2.1).



Graph 3 - Mean error for positions of the depth sensors

The distances examined are 2, 2.5, 3, 3.5, 4 meters and in each case to reduce the errors the mean error was calculated six times and the average of the six values was placed in the graph. The point clouds were merged on Kinect 0 that's why there are only two lines where blue is for Kinect 1 and orange is for Kinect 2. The best placement of the depth sensors was at 3 meters distance between them with mean errors for both kinects around 0.25cm, and the longest placement at 4 meters with around 0.42cm for both kinects. The calibration mean error almost doubled from 3 meters compared to 4 meters so camera placement plays quite a role in the quality of the merged point cloud.

In the graph below (see Graph 4), the three cameras were placed in a triangle shape (see Fig. 19) with a constant height of 1 meter, changing the distance between them, the time needed for calibration was calculated as explained previously (see chapter 6.1.2.1).

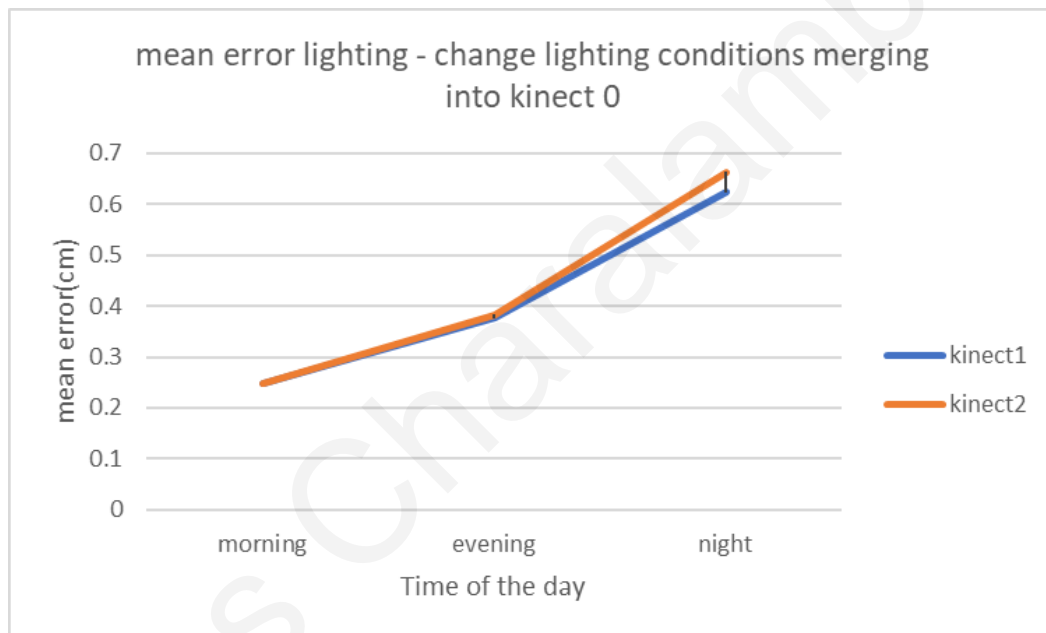


Graph 4 - time needed for calibration the depth sensors

The distances examined are 2, 2.5, 3, 3.5, 4 meters and in each case to reduce the errors the mean error was calculated six times and the average of the six values was placed in the graph. The point clouds were merged on Kinect 0 that's why there are only two lines where blue is for Kinect 1 and orange is for Kinect 2. And in this case, the best placement of the depth sensors is at 3 meters distance between them and the worst at 4 meters. The time at 4 meters is more than 2.5 times compared to 3 meters, since at 3 meters the time it took to do the calibration is about 1 sec for both cameras, while at 4 meters it is about 2.65 sec.

6.1.2.3 Calibration Mean Error for lighting conditions.

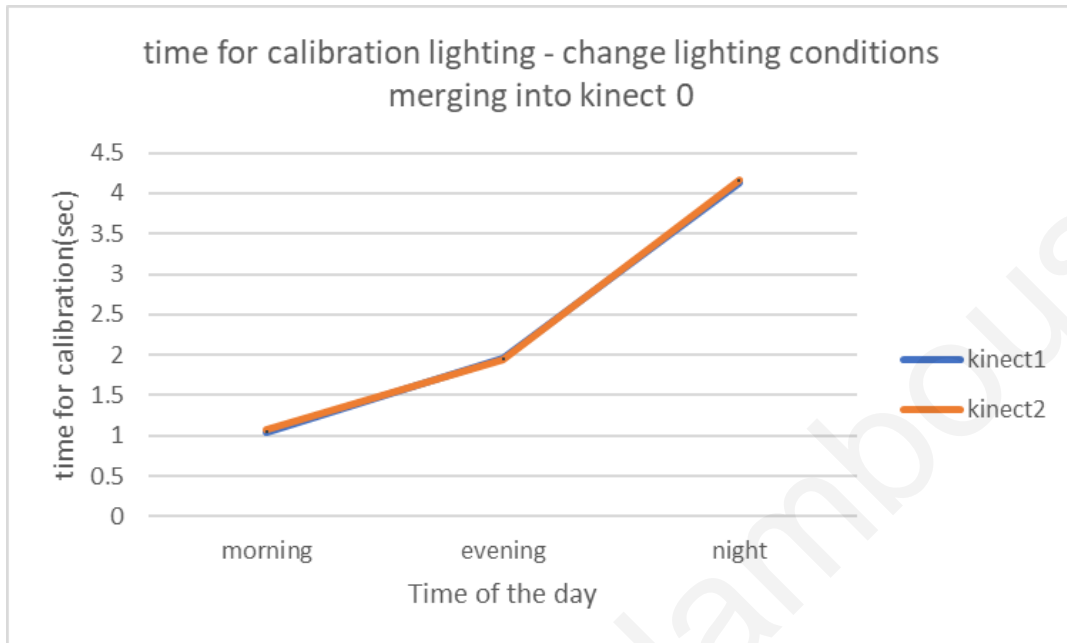
Another important factor that was examined to see if it affects the calibration and overall quality of volumetric capture is lighting. The depth sensors were placed 3 meters apart in a triangle shape with 1 meter height from the ground and the mean error and the time needed for calibration were calculated at three different times of the day. In the morning to midday when the light intensity was high, in the afternoon when the light intensity is moderate, and almost at night with low light intensity. In all three different conditions (morning - afternoon - evening) the mean error and the time were calculated six times and the average of the six values was placed on the graph to reduce the errors.



Graph 5 - mean error for lighting conditions

As shown in the graph above (see Graph 5) the intensity of the lighting plays a huge role. The blue line is for Kinect 1 and the orange line is for Kinect 2, where the point cloud merging was done on Kinect 0. In the morning the calibration means error for both Kinects was about 0.25cm, in the afternoon about 0.38cm, and in the evening about 0.65cm with Kinect 2 having a slightly larger mean error than Kinect 1. The calibration mean error increases non-grammatically since the difference between morning and afternoon is about 1.5 times and with evening the difference is about 2.6 times.

6.1.2.4 Time needed for Calibration - lighting conditions.



Graph 6 - time needed for calibration (lighting conditions)

As expected, the time needed to calibrate the depth sensors increases significantly as the intensity of the lighting decreases, as can be seen in the graph (see). The blue line is for Kinect 1 and the orange line is for Kinect 2. In the morning the time is about 1sec, in the afternoon it is about 1.8sec and in the evening about 4.2sec. And in this case it increases non-linearly since the difference from the morning compared to the evening was 1.8 times greater while the difference from the morning compared to the night was 4.2 times.

6.1.2.5 Time and Calibration Mean Error – Change height

Apart from the position of the depth sensors, the height was also examined, i.e. the distance between the cameras was kept constant at 3 meters by changing their height from the ground. Again, the depth sensors were placed in a triangle shape.

		Height(m)	Calibration Mean Error(cm)	Time Needed for Calibration(sec)
Set up 1	Kinect 1	0.7	0.306	1.17
	Kinect 2	1	0.492	2.63
Set up 2	Kinect 1	0.7	0.2467	1.03
	Kinect 2	1.3	0.683	3.4
Set up 3	Kinect 1	1	0.431	2.603
	Kinect 2	1	0.4367	2.647
Set up 4	Kinect 1	1.3	0.703	3.531
	Kinect 2	1	0.447	2.41
Set up 5	Kinect 1	1.3	0.692	3.56
	Kinect 2	1.3	0.710	3.61

Table 1 – Time and Calibration Mean Error - change height

Kinect 0 is fixed at height 0.7 and the other two point clouds will be merged into it. From the above table (see Table 1), 5 different scenarios were examined where in the third column you see the height that each depth sensor had from the ground and in the fourth column the calibration mean error. The results show that the height of the depth sensors clearly affects the final result. Since when the height of the Kinect is at the same height as the Main Kinect (Kinect 0 - 0.7m) then the Calibration Mean Error is approximately 0.25cm, when it is at 1 meter height the Calibration Mean Error is approximately 0.46cm and at 1.3 meters it is about 0.7cm.

As expected, the time it takes to perform the calibration depends a lot on the height of the Kinects. When the Kinects are at the same height as the main Kinect (Kinect 0 - 0.7m), the time they need to calibrate is about 1.1 sec, when they are at 1 meter about 2.5 sec and at 1.3 meters the time is about 3.6 sec.

6.1.2.6 NFOV vs WFOV

The depth sensors have two depth modes the NFOV (Narrow field-of-view depth mode) and the WFOV (Wide field-of-view depth mode), where the difference between the two is shown in the picture below (see Fig.38). NFOV provides a narrower field of view, which means it captures a smaller area but with more detail and a higher level of precision. WFOV, on the other hand, offers a wider field of view, allowing it to capture a larger area but with less detail and potentially lower precision compared to NFOV.

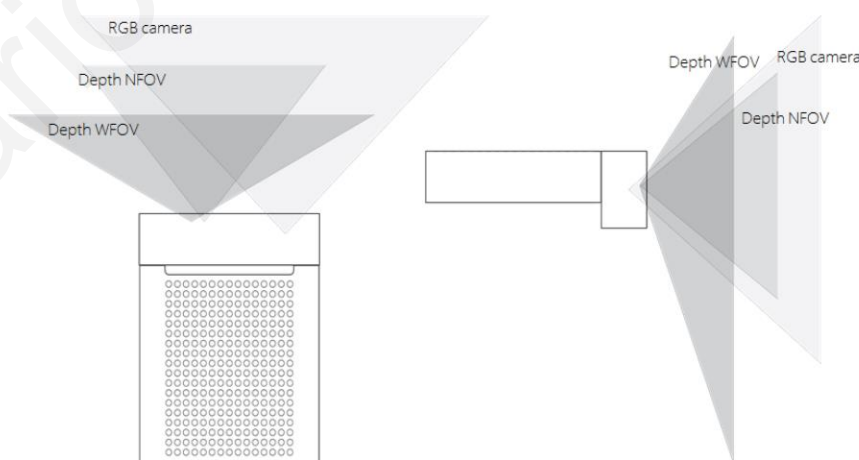
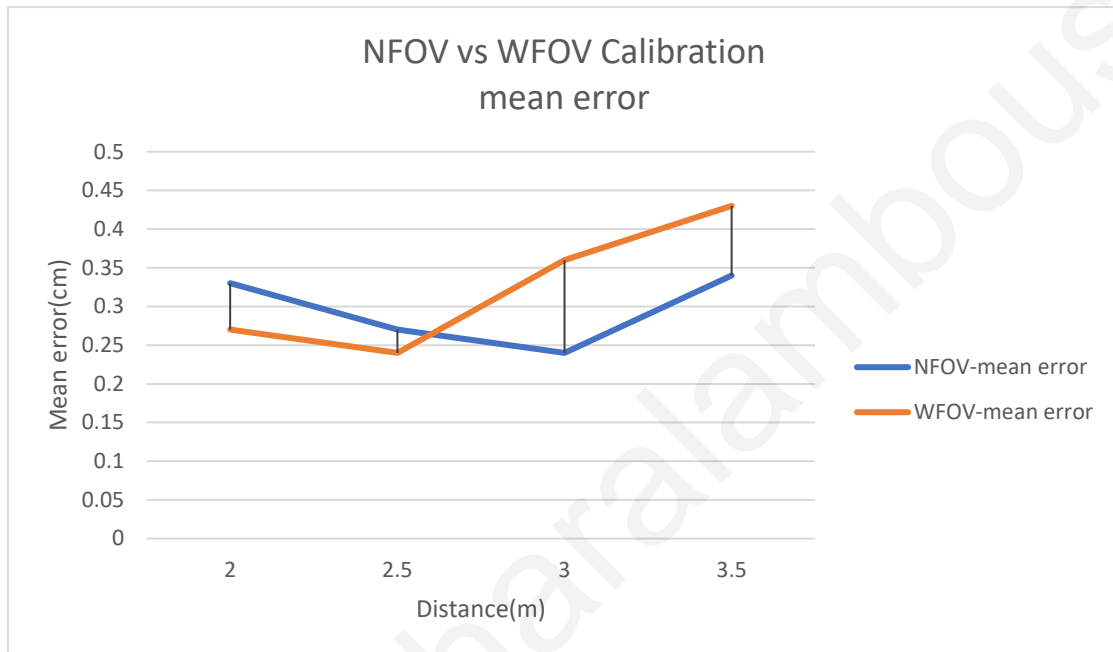


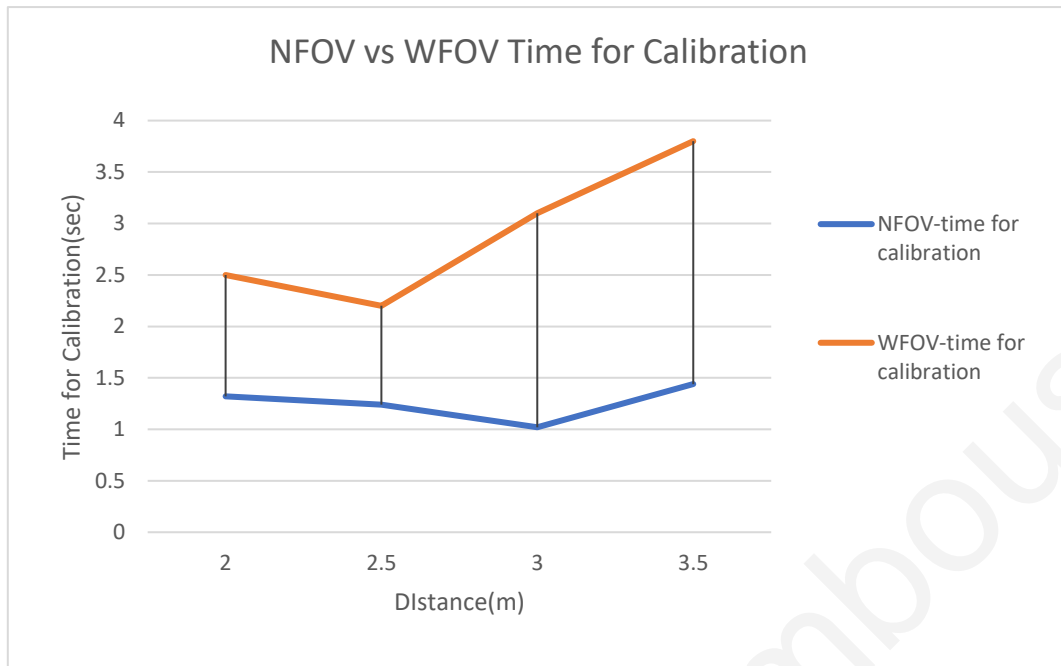
Figure 44 - NFOV vs WFOV

The depth sensors were placed at the same height from the ground (1 meter) and by changing the distance between them in a triangle shape the calibration mean error and the time needed for calibration were calculated for both depth modes. The distances considered are 2,2.5,3 and 3.5 meters and in each case six times were calculated and the average of the six values were plotted to reduce the errors.



Graph 7 - NFOV vs WFOV Calibration Mean Error

From the graph above (see Graph 7), the blue line is for NFOV calibration mean error and the orange line is for WFOV. It can be seen that for close distances WFOV has a slightly smaller calibration mean error with a better distance between depth sensors of 2.5 meters. In comparison, for longer distances NFOV is superior with a better distance between depth sensors of 3 meters.



Graph 8 - NFOV vs WFOV Time for Calibration

As before for the calibration mean error the time spent on calibration at each distance was calculated. In the graph above where the blue line is for NFOV time for calibration and the orange line for WFOV time for calibration it can be seen that WFOV takes much longer to do the calibration at all distances. This makes sense since it captures more space than NFOV and thus creates larger point clouds where it takes much longer to do the calibration. It took the least time the WFOV at a distance of 2.5 meters which was about 2.25 seconds and worst the distance of 3.5 meters with a time of about 3.75 seconds. Contrast this with the NFOV where it was best at 3 meters with a time of about 1 second and worst at 3.5 meters with a time of about 1.5 seconds. The time difference needed for the calibration is quite large where the WFOV is 2.5 times slower than the NFOV.

6.1.2.7 Ball-Pivoting vs Poisson Algorithm

As mentioned before, the only parameters that affect the ball-pivoting algorithm are the radius that the ball will circle to create the surface triangles and the maximum neighbours it will search in this space.



Figure 45 - Point Cloud

in the image above(see Fig.45) is the Point cloud tested to mesh with the two algorithms.



Figure 46 - Mesh radius=2 max_nn=30

Changing these two parameters creates the mesh. In the adjacent image (see Fig.46) with radius=2 and maximum neighbours 30 the result is bad with too many holes and low detail mesh. The reason for this is that not enough triangles were created and so it has too many holes.



Figure 48 - Mesh radius=5, max_nn=90

In the adjacent image(see Fig.48) with a radius of 5 and a maximum neighbor of 90, the result is better with better mesh detail. However, there are still several holes because not enough triangles were created, even in the case of Pally.



Figure 47 - Mesh radius=10, max_nn=300

the last mesh(see Fig.47) made was with a radius 10 and maximum neighbors 300 and the result is much better with higher detail and fewer holes. The conclusion is that you create a better mesh as you increase the radius and the neighbors you can find within that space.

The second algorithm(Poisson) the parameter that affects the quality of the mesh is the depth of the tree. The greater the depth, the better the mesh will be created. Again from the same point cloud (see Fig.45) they made the meshes by changing the depth.



Figure 51 - mesh depth = 6



Figure 49 - mesh depth = 8



Figure 50 - mesh depth = 10

In the above images (see Fig 49-51) , the difference in the quality of the mesh by changing the depth is huge since initially with depth = 6 the result is miserable with very few triangles being created while as the depth increases the result is much better. Better results were obtained with the Poisson algorithm since the meshes had no holes and fewer artifacts than the meshes created with the ball-pivoting algorithm.

6.2 Future Work

Further improvements will always be made and certainly the first is to improve the quality of volumetric capture, developing better algorithms for capturing and reconstructing 3D models, improving the accuracy and resolution of the captured data.

Also another improvement is to be able to make the 3D model in real time efficiently and with good quality. For this it is necessary to have the appropriate hardware that can support this complex process.

And finally, to make different applications such as VR and AR applications to create more immersive and interactive experiences for users, allow users to interact with the captured data in real-time, such as by enabling them to move around or manipulate the captured objects.

7 Bibliography

- [1] URL : <https://www.unrealengine.com/en-US>
- [2] URL : <https://azure.microsoft.com/en-us/products/kinect-dk>
- [3] URL : <https://pointclouds.org/>
- [4] URL : <https://github.com/microsoft/Azure-Kinect-Sensor-SDK>
- [5] [URL:https://medium.com/keentools/keentools-facebuilder-x-metahuman-guide-81bc193ef2a](https://medium.com/keentools/keentools-facebuilder-x-metahuman-guide-81bc193ef2a)
- [6] URL : <http://www.open3d.org/docs/0.6.0/#>
- [7] URL : <https://opencv.org/>
- [8] "Free-Viewpoint Television" by T. Fujii et al., Proceedings of the IEEE, 2012. This paper provides an overview of free-viewpoint television, a technique used in volumetric capture to create immersive 3D content.
- [9] "State-of-the-art in 3D Reconstruction with RGB-D Cameras" by C. Stachniss et al., IEEE Transactions on Robotics, 2014. This paper discusses the state-of-the-art in 3D reconstruction using RGB-D cameras, which are commonly used in volumetric capture.
- [10] "Volumetric Performance Capture from Minimal Camera Viewpoints" by P. Pérez et al., ACM Transactions on Graphics, 2018. This paper presents a method for volumetric performance capture that requires only a few camera viewpoints.
- [11] "Live Volumetric Performance Capture" by M. Zollhöfer et al., ACM Transactions on Graphics, 2018. This paper presents a system for live volumetric performance capture, which can be used to create real-time 3D content.
- [12] "A Survey of 3D Modeling Techniques for Interactive Applications" by D. Luebke et al., IEEE Computer Graphics and Applications, 2003. This paper provides an overview of various 3D modeling techniques, including volumetric capture, and discusses their applications in interactive environments.
- [13] "Volumetric Capture of Humans with a Single RGBD Sensor" by J. Xie et al., IEEE Transactions on Visualization and Computer Graphics, 2019. This paper presents a method for volumetric capture of humans using a single RGBD sensor.
- [14] "Efficient Volumetric Video Coding for Interactive Applications" by H. Aksay et al., ACM Transactions on Multimedia Computing, Communications, and Applications, 2020. This paper presents an efficient method for compressing and transmitting volumetric video data, which is important for interactive applications.

- [15] RUSU R. B., COUSINS S.: 3D is here: Point Cloud Library (PCL). In IEEE International Conference on Robotics and Automation (ICRA) (Shanghai, China, May 9-13 2011)

Marios Charalambous

Appendix

7.1 Source Code

Content of Source Folder :

- PointCloud - the textures created by capturing(depth and color image)
- TXTfiles – all files stored
- Blueprints – All blueprints
- Animations – Metahuman animations
- Maps – Metahuman and Volumetric capture scenes
- Metahuman – Body, Face, Textures of my metahuman
- Scripts – C++ files
 - ◆ AzureKinect.cpp – Initialize and start Kinect.
 - ◆ PointCloudRender.cpp – Render Point cloud in the scene
 - ◆ RegistrationManager.cpp – Get all the result from registration
 - ◆ TXTManager.cpp – check txt file
 - ◆ RecordPointCloud.cpp – Save positions and color of all points
 - ◆ SaveRegistration.cpp – Save transformation matrixes for calibration
 - ◆ GridDetector.cpp – detect Grid for Calibration
 - ◆ MultiPointCloudRegistration.cpp – Perform calibration for all kinect
 - ◆ Structs.h - contains two structs which are used as input and output for the lattice detector
 - ◆ Vector4f.h - is our internal Vector class.
 - ◆ Matrix4f.h - is our internal Matrix class.

7.2 Structure

