Fachbereich 3 - Mathematik & Informatik

Computer Graphics and Virtual Reality group (CGVR)

Hand mesh construction from poses using spiral convolution in Unreal Engine

Thomas Höring
matriculation number:
4381005

Master Thesis
for attainment of the academic degree of

Master of Science
(M. Sc.)

in Computer Science

Examiner 1: Prof. Dr. Gabriel Zachmann
Computer Graphics and Virtual Reality

Examiner 2: Dr. Thomas Röfer
Multi-Sensor Interactive Systems

Advisors: Prof. Dr. Gabriel Zachmann
Janis Roßkamp

Bremen, 19th March, 2024

Declaration of Authorship

I hereby declare that I am the sole author of this master thesis titled "Hand mesh construction from poses using spiral convolution in Unreal Engine". Any use other materials and works of other authors is properly acknowledged at their point of use. I further declare that I have not submitted this thesis at any other institution in order to obtain a degree.

Bremen, 20.03.2024	Thomas Höring
Place, Date	Thomas Höring

Danksagung

Zunächst möchte ich meinen Prüfern Gabriel Zachmann und Thomas Röfer für ihr Interesse an diesem Thema und ihrer Betreuung danken. Besonderer Dank geht an Janis Roßkamp, der sich über den gesamten Verlauf der Arbeit Zeit für genommen hat, um Lösungsansätze zu besprechen und Feedback zu geben, von denen ich viel lernen konnte. Danke auch an alle weiteren Mitgliedern und Studierenden des CGVRs, die mit Fragen und Anregungen oder im Hintergrund, bei dieser Arbeit unterstützt haben.

Außerdem möchte ich mich bei meinen Freunden und Freundinnen bedanken, die immer ein offenes Ohr und Anregungen hatten. Dabei haben sie mich nicht nur seelisch, sondern auch beim Korrekturlesen unterstützt. Besonderer Dank gilt hier Lisa, Lukas und Jan.

Nicht zuletzt möchte ich meinen Eltern danken, die mich nicht nur während dieser Masterarbeit, sondern über das gesamte Studium hinweg stets unterstützt haben. Ohne ihre Ideen, Motivation und die Ruhe abseits des lauten Bremens, wäre nicht nur diese Masterarbeit anders verlaufen. Danke, dass ihr diese Zeit ermöglicht habt.

Abstract

The virtual reality field is continuously developing with recent advances in entertainment, medicine, academia, and business in general. The representation of virtual avatars thereby becomes as important as never before. There is a special focus on the visualization of hands, as they are in view most of the time, are our main tool for interaction with objects, and are vital for non-verbal communication. However, their high degree of complexity makes them hard to simulate realistically. A way to create anatomically realistic hands quickly is in demand.

This thesis presents the implementation, training, and deployment of a neural network for the construction of 3D hand meshes from poses in Unreal Engine 5.
The approach is based on a variation of the spiral convolution, a method to establish a local neighborhood of vertices on the surface of a mesh. These vertices form a kernel on which a convolutional neural network can operate. This allows the creation of fast neural networks that operate on a 3D mesh. I use this functionality to train the network on a realistic hand model generated with the NIMBLE project. The trained network is deployed in Unreal Engine 5, which, in combination with a Leap Motion Controller, enables real-time hand tracking and construction with the detail level of NIMBLE hands.

A spiral convolution approach was implemented and compared to a previous variant. In a hyperparameter study, different architectures and approaches were tested before training a network that constructs the 3D mesh directly from the hand pose. Visual comparison with ground truth data shows an excellent approximation of the original NIMBLE model in a fraction of the time. Construction of NIMBLE-like models in real-time is thereby made possible.

Contents

1 Introduction **1**

2 Related Works **3**
2.1 Hand Models . . . 3
2.1.1 MANO . . . 4
2.1.2 NIMBLE . . . 4
2.1.3 Ultraleap Hand Tracking model . . . 4
2.2 Convolutional Neural Networks . . . 5
2.2.1 Convolutional layer . . . 6
2.2.2 Pooling Layer . . . 6
2.3 Unreal Engine . . . 6
2.4 Spiral Convolutions . . . 8
2.5 Mesh construction from poses . . . 9

3 Method and Implementation **10**
3.1 Training Data . . . 10
3.1.1 NIMBLE . . . 10
3.1.2 Leap Motion . . . 11
3.1.3 Input Normalization . . . 13
3.2 Adaptation from SpiralNet++ . . . 14
3.3 Spiral Convolution . . . 14
3.3.1 Spiral Kernel . . . 15
3.3.2 Hungarian Method . . . 17
3.3.3 Starting Point . . . 18
3.4 Pooling . . . 18
3.5 Network Architecture . . . 20
3.6 Unreal Engine Integration . . . 23
3.7 Python Server . . . 23
3.7.1 Neural Network Serialization . . . 24
3.7.2 LibTorch for Unreal Engine . . . 25
3.7.3 Hand tracking . . . 26
3.7.4 Data Loading and Processing . . . 28
3.7.5 Vertex Distance Shader . . . 28

4 Analysis **30**
4.1 Comparison with SpiralNet++ . . . 30
4.2 Parameter Study . . . 33
4.2.1 Study Setup . . . 33
4.2.2 Results of the parameter analysis . . . 35
4.3 Pose Encoder . . . 40
4.4 Tracking . . . 45

5 Discussion **48**
5.1 Comparison on CoMA . . . 48
5.2 Parameter study . . . 48
5.3 Pose Encoder . . . 49
5.4 Limitations . . . 50
5.4.1 Reproduciblility . . . 50
5.4.2 Computing Power . . . 50
5.4.3 Training data . . . 50

5.4.4 Mapping 50

6 Conclusion **52**
6.1 Future Work 53

References **54**

A Appendix **i**
A.1 Content of memory stick i
A.2 Results of the parameter study ii

List of Figures

1 Examples for MANO, NIMBLE, and Ultraleap . . . 5
2 Screenshot Unreal Engine . . . 7
3 Joints in NIMBLE . . . 13
4 Spiral and Ring Spiral approach . . . 16
5 Architecture of autoencoder . . . 21
6 Architecture of pose encoder . . . 22
7 Overview Unreal implementation . . . 25
8 Leap Motion output . . . 26
9 CoMA comparison overview . . . 32
10 CoMA loss curve . . . 33
11 Learning rate influence . . . 36
12 Sequence length influence . . . 37
14 Parameter count and inference time . . . 37
13 Network depth influence . . . 38
15 Sequence length influence . . . 38
16 Output channels influence . . . 39
17 Latent space with '128' . . . 40
18 Pose encoder error . . . 41
19 Pose encoder loss . . . 42
21 Histogram of slow and fast executions . . . 42
20 Pose encoder inference times . . . 43
22 Visual comparison with pose encoder . . . 44
23 Visual comparison overview . . . 45
24 Ghosting effect of procedural mesh . . . 46
25 Comparison with tracking . . . 47
26 Pose with mapping problem . . . 51

List of Tables

1 Results: Comparison with SpiralNet++ . . . 31
2 Results of the parameter study for autoencoders . . . ii
3 Results of the parameter study for pose encoders . . . v

1 Introduction

The conversation around Virtual Reality has reached a new peak. Consumer products from Meta and, more recently, Apple push the industry forward and can capture public attention. Outside the consumer market, which is mainly focused on VR entertainment, the technology is used in 3D modeling, training, healthcare, and academia. As more and more people come into contact with virtual reality, concepts of virtual presence, as the simulation of being part of a virtual environment, and immersion in those environments become eminent [16, 66]. VR systems reach for a high degree of immersion as it usually improves the VR experience, which in many cases is a question of graphical fidelity and a self-concept a user has about himself or herself [16, 40]. A constant part of this discussion is the representation of hands. Not only are they our main tool for manipulating our environment, but they are also essential for non-verbal communication [21]. Body language is vital to human interaction and should not be underestimated. Additionally, the visualization of hands is important in VR because they are in view most of the time and are often the only part one can see of one's own character. This importance creates a demand for realism in hands and motivates the focus on hands in this thesis.

However, as most artists and animators know, 'Hands are hard.' The versatility our hands allow us, also make them very complex. Twenty-seven bones and numerous joints with various degrees of freedom allow for a large variety of poses. Bones, muscles, and tendons move the skin and create folds and mounts that are very difficult to visualize [62]. Therefore, Meta uses a basic and video-game-like visualization, while Apple tries to avoid using virtual hands altogether. Simulation approaches like those do not capture the intricacies of human hands and fail to recreate a realistic-looking mesh. While hand models exist that can create hands at a very realistic degree from a hand's pose, those models are typically very slow. NIMBLE is such a model: It is based on MRI scans and depth images of hands and simulates the underlying bones and muscle groups of a hand. As those simulations are based on physical constraints and complex optimization processes, their computation takes too long for real-time applications like VR [35].

This thesis wants to fill the gap between real-time applications in need of quick models and resource-intensive, realistic hand models, allowing for 3D meshes of hands closer to physical precision in real-time. To reach this goal, I implement a convolutional neural network and train a model that approximates the hand model NIMBLE. This model is deployed in Unreal Engine 5 to be used in a real-time context. The neural model receives the pose of a hand specified as joint angles and outputs the vertex positions of a hand in that specific pose.

The neural network I implement is motivated by the success of Convolutional Neural Networks (CNNs), which are normally used for neural image processing. The series of research that this work follows utilizes spirals to extend the functionality of CNNs to 3D data. While classic CNNs for images create a kernel from pixels around a pixel as a grid, finding an equal number of neighbors on the surface of a 3D mesh is less trivial [48]. Spirals around a vertex of a mesh define a neighborhood of that vertex, which is utilized as the kernel of that specific vertex. For the implementation of this idea, previous work by Gong et al. [22] is used as a base. Chentanez et al. [9] propose changes to that model to overcome the limitations of that model and offer new ideas regarding the spiral generation and pooling approaches. I implement the algorithms described and test them against the implementation of Gong et al. [22] to evaluate the effect of the proposed changes. This way, I hope to provide an implementation that can match the original results and improve on the implementation of Gong et al. [22], thereby moving the field of spiral convolutions forward, independently of hand simulation.

In pursuit of the creation of 3D hand meshes, I utilize the NIMBLE hand model to generate data from hand poses. These poses are recorded with a Leap Motion Controller. The dataset

created this way is used to train an autoencoder on meshes. The encoder is then exchanged with a small, fully connected encoder that translates the pose of the hand to the semantic representation learned by the autoencoder. This creates a neural network that can generate a 3D mesh from the pose of a hand in a short time. To ensure good performance of the neural network, I perform a parameter study, evaluating various hyperparameters and network structures.

For use in a real-time application, I use the Unreal Engine 5, a modern game engine with support for VR and a high degree of customizability. I integrate LibTorch in the Unreal Engine as a custom plugin, which allows me to load and execute a serialized model of the neural network I trained beforehand. Integration of the Leap Motion Controller allows hand tracking, whose data can be transferred to the neural network. The final output is a 3D model of a hand, similar to one created using NIMBLE, but regressed in a fraction of the time. It is deployed in a capable game engine and is ready to be used for various projects.

Overview

In the following chapter, *Related Works* (2), I will present work adjacent to the topics in this thesis. First, relevant hand models are presented. Next, a foundation for Convolutional Neural Networks and the Unreal Engine is set. The core of the related work are spiral convolutions that define how meshes are processed with the neural network. Lastly, I mention work that focuses on the construction of meshes from poses utilizing other techniques.

In the chapter *Method and Implementation* (3) the main ideas of this thesis are laid out. It includes the full implementation of a neural network, starting with the collection of training data, the implementation of the spiral convolution, and the new pooling approach. This concentrates on the novel ideas of Chentanez et al. [9]. Describing the network architectures used in the project concludes the implementation of the neural network. The chapter closes with the network's integration into the Unreal Engine 5. Network serialization dominates this part, with decisions and issues about it outlined.

The *Analysis* (4) features a comparison with the SpiralNet++ on the CoMA faces data, to test the new implementation. In a hyperparameter study, the best parameters and architecture for an autoencoder within the given architecture framework are found. As those results are necessary for the training of the network able to encode poses, they are discussed shortly. It follows an analysis of the pose encoder, which is connected closely with real-time tracking.

In the subsequent chapter, *Discussion* (5), the remaining results of the analysis are discussed. The main focus are the results of the pose encoder and the tracking as they represent the final work of the thesis. Limitations of the project and the model itself are part of the discussion as well.

Finally, the *Conclusion* (6) summarizes the path and the achievements of this work while providing how future work might improve on it.

2 Related Works

The related work for this thesis spans wide, as many components and technologies come together to be used in unison. While I consider the most relevant related work publications regarding spiral convolution and its development, the history and development of fundamental technologies will be covered first.

These fundamentals are divided into three chapters: hand modeling, convolutional neural networks, and the Unreal Engine. For each, I present relevant topics and research that give an overview of the field and allow for better understanding when making decisions later in this thesis.
After presenting the fundamentals, I will review the two primary papers my research is based on and examine the line of work that leads to them. How do they advance the field I am working in, and how are they relevant to this paper? Lastly, I will shortly expand on papers with a similar goal of constructing 3D meshes using neural networks but do so using different approaches.

2.1 Hand Models

The modeling of hands for virtual environments is a problem that has existed since the start of animation. In animation, characters often have just four fingers to make drawing and animating them easier [43]. Since 3D video games with realistic graphics exist, and at the latest, with the rise of virtual reality games and applications, the modeling and animation of realistic hands have become highly relevant. Fast and accurate modeling of the user's hand becomes crucial, especially for virtual reality systems. Hands are not only a tool for interacting and manipulating but are also crucial for communication as a vital part of body language. While virtual reality and communication through virtual reality systems gain more traction, this issue is more relevant than ever [65].

For those applications, hand models were developed to create a digital representation of a hand. Many of those hand models are parametric, meaning that they feature several parameters that allow the posing or deformation of the hand. The number of parameters can differ from model to model. Most essential here is the pose of the hand, which needs to be encoded into those parameters. The joints and their angle are typical for those parameters; others may describe the shape of the hand. The number of parameters is typically reduced to obtain a model that is as simple as possible. For example, the fact that the joints of the fingers cannot move in all directions in the same way is utilized. The number of degrees of freedom can thereby be decreased, and the model is simplified. What is interesting here is that the thumb is often represented differently compared to the other four fingers. Studies in biomechanics showed that the movement of the thumb is much more complex [26]. That leads to it being typically defined with more degrees of freedom than other fingers and a different implementation between hand models. This can lead to problems when comparing or transferring information between them.

The output of a parametric hand model does not only have to be a 3D model of a hand but can take many forms. Some models include the arm or the texture of the hand [39]; others focus on the inner workings of a hand, like Li et al. [36], who create anatomically and physically precise representations of the bones of the hand.
I discuss three hand models, the first two of which are parametric. The second one, NIMBLE, is largely involved in generating training data. The third model is *Leap*, which is involved in hand tracking and training data generation later in this thesis.

2.1.1 MANO

MANO is a hand model from the paper "Embodied Hands: Modeling and Capturing Hands and Bodies Together" [54]. The authors of this paper address the inherent challenges that arise when dealing with hands. Especially when scanning a whole body, hands become complex, as they are relatively small and often occluded, e.g., by other parts of the hand. They developed the hand model MANO (hand Model with Articulated and Non-rigid defOrmations) to overcome problems like low resolution, occlusion, and noise when capturing the shape and pose of hands in three dimensions. They aimed to create a realistic but low-dimensional model that can be used for all hand shapes and poses. The model "captures non-rigid shape changes with pose" and utilizes linear blend skinning [54, p. 1]. To train the model, 31 subjects provided about 1,000 3D scans of their hands in various poses. The paper combined the MANO model with a body shape model (SMPL) to create a high-quality full-body scan. However, MANO seems to be getting more attention in the scientific community. Even with newer hand models and improvements available, like Chen, Wang, and Shum [8], it still is relevant and used in recent papers like Huang et al.[27].

2.1.2 NIMBLE

NIMBLE is a hand model developed in 2022 by Li et al. [35]. It is a parametric hand model built on MANO but goes further by appreciating the complexity of the inner workings of the hand and trying to simulate them. Instead of 3D scans of hands, they use MRI scans provided by Li et al. [36] for the PIANO model, which allows them to annotate bones, muscles, and skin. Those can later be used for simulation, in which multiple 3D objects are used to simulate the bones and muscle groups. With this approach, they aim for a high level of realism [35]. They achieve this by setting various kinetic rules in which deformation is allowed, thereby mimicking natural hand behavior. Hands are deformed using linear blend skinning, with localized skinning blend weights to only deform relevant parts. The muscle simulation allows for muscle stretching and bulging, affecting the skin simulation. The NIMBLE model also produces albedo and specular textures as well as a normal map. They add to the realism of the hand once through their coloring, but especially with the visualization of folds and small wrinkles, for which the mesh is too coarse [35]. This creates a model of high fidelity with shape blend shapes and pose blend shapes.

2.1.3 Ultraleap Hand Tracking model

The Leap Motion Controller is a proprietary hand-tracking device distributed by Ultraleap Limited. It is used for hand tracking in this thesis. More about the tracker is explained in 3.7.3. The Leap Motion Controller's tracking and processing algorithms, including the hand model they integrate into their software, are not public. From the visualizations, the rigging process, and the documentation they provide, they seem to use a skeletal-based hand model instead of a parametric one [38]. In a skeletal model, bones are created in a hierarchical structure. The movement of each bone influences the bones further down in the hierarchy, and the mesh is weighted to follow one or multiple bones. The mesh can be animated by moving the bones. That representation of the hand can be transferred into parameters for NIMBLE. Implementation details, especially regarding the thumb, can nonetheless lead to complex relationships between models. The documentation of the Leap Motion states that, even though the thumb anatomically has no intermediate phalange, they use it in their model and compensate for this with a zero-length metacarpal bone [38]. Differences like these complicate comparing and working with hand models, especially when the details of the hand model are treated as a trade secret.

Figure 1: Examples for MANO, NIMBLE, and the Ultraleap hand model, from left to right. The pose is not exact for all hands, as it is estimated from the MANO pose and then recreated with real-time tracking. There is a clear improvement in fidelity from MANO to NIMBLE. The Ultraleap model looks very basic and even simpler than the MANO model.

2.2 Convolutional Neural Networks

The idea of neural networks has existed for quite some time in computer science but gained a lot of traction in the last two decades, as more computing power allowed the training of large networks. By then, many forms of layers used in neural networks and architectures were introduced to serve their specific purpose. One of those layers is the convolutional layer, which made a lasting impression on the field of machine learning.

Shortly before the turn of the millennium, Convolution Neural Networks (CNNs) came into existence. They showed that it is possible to perform image processing with small neural networks and without image preprocessing. In the frequently cited paper "Handwritten Digit Recognition with a Back-Propagation Network", the authors use a CNN for the first time to recognize handwritten numbers from zip codes, the MNIST dataset [34].

While CNNs were first used to recognize handwritten digits, their potential was soon recognized. Their ability to reduce the complexity of an image to a very high degree compared to feedforward networks makes them suitable for processing complex imagery. Classic image processing algorithms rely on recognizing basic shapes and structures but are typically unable to learn and must be actively programmed. With breakthroughs in the domain of image classification with models like AlexNet [32], CNNs were adapted for other tasks as well like segmentation and image creation [29].

Since then, they have also been used in other applications like sequential data, e.g., audio or graphs. In those cases, relations between input features are different compared to pixels of an image, but methods to craft a similar relation between them can be found [25, 44].

The centerpiece of a convolution neural network consists of two types of layers: the convolutional layer and the pooling layer. A brief explanation of their functionality follows.

2.2.1 Convolutional layer

A fully connected layer in a neural network combines each input feature with each output feature, hence the name. The influence behind every connection needs to be trained, which can lead to very long training times if input and output reach a large size of features. A fully connected layer sees all inputs as independent of each other and has to learn relations between them with a lengthy training process. The convolutional layer promises to tackle this disadvantage by exploiting relations in the input data. Convolutional Neural Networks are mainly used for image processing. The features of images are their pixels, but they are not independent of each other. Neighboring pixels also have a semantic relation to each other. Many pixels together may form shapes or belong to the same object in a photo.

CNNs use the knowledge about relationships in the data to learn kernels that describe a feature of a close neighborhood. In the context of image processing, a kernel works like a sliding window that iterates over all pixels of the image. Typical kernel sizes are 3×3, 5×5, or 7×7 pixels. The relationship between those pixels is learned, and their effect is shared over all pixels of the image. The idea behind this is that aspects of the image that are relevant in one region are also relevant in other parts of the image. A basic kernel, for example, might learn to recognize horizontal lines. Combined with other kernels that learn different input features over multiple layers, the ability of the convolutional layers increases. A kernel, traversing over the pixels of an image, performs a convolution operation by computing the dot product between the kernel values and the pixel with its immediate neighborhood. If the result of that operation exceeds a certain threshold, the output of the convolutional layer is activated. The number of different kernels is also described as the depth of the layer or the channels of the layer. Each is responsible for detecting a specific feature or pattern in the input data.

A disadvantage convolutional layers have compared to fully connected layers is that they do not have the overview of the whole image because they can only work on the immediate neighborhood. To counteract this, pooling layers are introduced into CNNs.

2.2.2 Pooling Layer

The pooling layer is an essential tool of the CNN. As the name suggests, pixels of the input data are pooled together. In a 2×2 pooling layer, four pixels of the input are combined into one pixel in the layer's output, effectively reducing the resolution of the image. Larger sizes of pooling fields are also possible. There are two main variants of pooling procedures: Maximum Pooling and Average Pooling. Maximum Pooling replaces the pooled values with the maximum of those values. Average Pooling forms the average of the values in question.
This way, the input of the next convolutional layer becomes smaller. The kernel's receptive field can then perceive a larger area of the image relative to the original image. It enables the network to recognize larger patterns and objects in high-resolution images. It is typical for the architecture of a CNN to decrease the image resolution with pooling layers while increasing the number of channels of the convolutional layers. The goal of this approach is to stabilize the complexity of each layer.

2.3 Unreal Engine

The Unreal Engine is a game engine developed by Epic Games. It is not only used for game development but also for design work in architecture and the automotive industries, as well as for virtual production or animation and rendering work.
It gained popularity not only for its business model that allows for free use in non-commercial applications and low-revenue-projects, which makes the engine relevant for new game developers and small studios but in this context, more importantly, academic research [56].

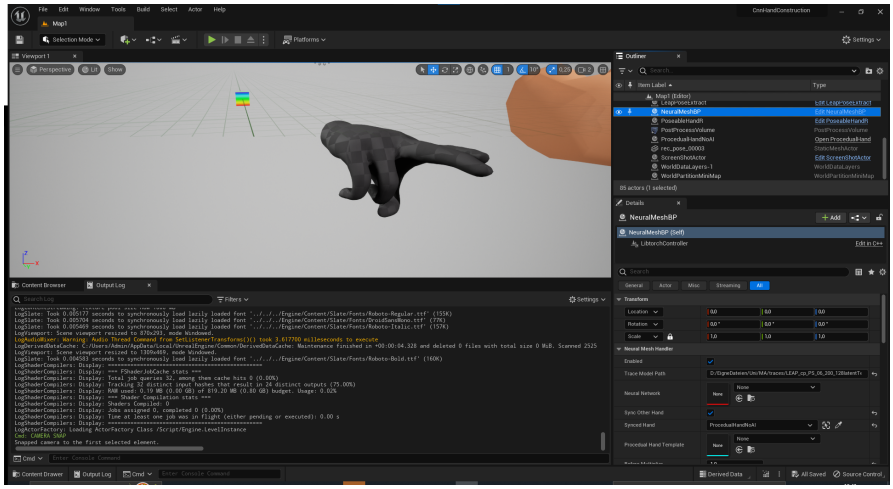


Figure 2: Screenshot of the Unreal Engine. The project of this thesis is open, and key components of the editor are visible.

Modern game engines are very complex pieces of software that are often proprietary to a game development studio. Popular freely available alternatives that are free to use non-commercially are Unity and Godot. Unreal Engine stands out through its high performance and feature richness, allowing for realistic graphics and complex systems in the final product. The high performance is especially interesting for virtual reality projects, as slow refresh rates can lead to motion sickness, leaving users nauseated.

Similar to other game engines, the features of the Unreal Engine are highly focused on real-time rendering. Their framework supports efficient 2D and 3D rendering, lighting systems, audio rendering, handling of controls and inputs, physics handling, collision management, animation, and networking. The engine is constantly being improved, and features are being added or evolved. The last major version was released on the 5th of April 2022, which added, among other things, a global lighting system and a new render concept to enable fast rendering of scenes with high triangle counts. The current version is Unreal Engine 5.3. With version 5.0, Epic Games added the “Neural Network Inference” plugin, which allows inference of serialized neural networks. Previously, this functionality was available through external plugins. This plugin enables the execution of most neural networks directly within the engine. Examples of its usage are the experimental “Machine Learning Deformer” plugin, which helps to create more realistic skin deformations for skeletal meshes, or a network that performs a style transfer of the rendered image.

Despite Unreal Engine’s many functionalities, it is relatively beginner-friendly, especially when experienced with other 3D software or game engines. Many example projects are presented when creating a game, which can be a good starting point for most games. UE5 also implements a graphic programming language, the Blueprint system. Users who are uncomfortable with the underlying C++ can code visually, even though the system is limited, slower, and less flexible than writing efficient C++ code. The communication between the Blueprint and C++ systems, however, is limited.

The community, the asset store, and the properties of the engine allow a wide variety of developers to make games, which in return leads to a more significant number of visually impressive games published through small studios or single developers. Epic Games also developed popular games, like Fortnite, which further contributed to the engine’s popularity.

2.4 Spiral Convolutions

This thesis focuses on a series of works implementing a graph-based convolutional neural network. Forming a kernel on a graph or mesh, similar to a kernel for a CNN working with images, is way more ambiguous. Therefore, a wide variety of approaches are open for discussion. A kernel needs to create an orientation and order and establish a local neighborhood on the surface of the mesh. Before the spiral convolution approach, creating a kernel for the convolution relied on patches or graph-based approaches [37]. Those patches are placed on the surface of the mesh and represent an area determined before by geodesic sampling. The patches are then used for learning in neural networks similar to CNNs [5, 42, 44]. While their results were good, the mesh representation was complicated and required a resampling. Graph-based approaches extend the basic properties of their domain to the features of meshes [31, 60].

Lim et al. [37] is mainly concerned with a simple representation of 3D meshes that is less complex and resource-intensive than previous approaches that rely on resampling of the mesh. They first present the approach of a spiral kernel. The spiral is formed by moving from a center vertex to a vertex next to it and then around the center vertex in a spiral. The vertices of the spiral are saved as a sequence for the specific center vertex. This spiral kernel creates order and provides a local neighborhood on the mesh. Even though Lim et al. [37] does not use CNNs, but Recurrent Neural Networks (RNNs) in the form of their LSTM-NET, they proved the viability of spirals to represent meshes. RNNs are neural networks that can process sequential data of varying sizes; Long Short-Term Memory networks (LSTMs) are a special case of an RNN. They highlight the simplicity of the method, the small amount of resampling, and state competitive results compared to other patch-based methods. As possible future work, they mention the slower training of LSTMs and fully connected layers and propose one-dimensional convolutional layers [37].

Bouritsas et al. [6] use this spiral representation first to create a CNN to synthesize meshes. Similar to this work, they also work with deformable meshes, more specifically of faces and full body meshes, based on the FAUST, CoMA, and Mein3D datasets [3, 4, 52]. Their results showed a significant improvement in the approach of Lim et al. [37]. This can be attributed to the use of a CNN architecture and fixed spirals over different meshes and epochs. They also introduce the idea of starting a spiral at a fixed point, geodesically closest to a fixed vertex. Their autoencoder structure uses convolutional layers with smaller kernels followed by pooling layers. For their pooling, they use a method inspired by Ranjan et al. [52], where faces are extrapolated using barycentric coordinates.

In SpiralNet++ by Gong et al. [22], the idea of the fixed starting point is rejected. They generate spiral sequences only once for all meshes and introduce a new “diluted spiral”. The diluted spiral only uses every n-th vertex of a spiral, creating a greater local neighborhood while keeping the length of the kernel the same. Compared with the Neural3DMM network of Bouritsas et al.[6], they achieve higher accuracies and have much faster compute times on the CoMA dataset. Results on the FAUST dataset may be more comparable to the deformation of hands than CoMA because of the variety of strength of articulation, which is much higher on bodies with limps and hands than on faces. SpiralNet++ elaborates on their pooling function:

They approximate the surface error using quadratic metrics. Based on those errors, edges are chosen for an edge collapse iteratively. The edge contractions stop when a specific downsampling factor is reached. For their studies, they use a downsampling factor of four. A transformation matrix between the regular and downsampled mesh is created. Pooling during the execution of the neural network can thereby be reduced to a simple matrix multiplication. Unpooling is again done by “using the barycentric coordinates of the close[s]t triangle in the decimated mesh” [22, p. 6], based on the work of Ranjan et al. [52]. They first proposed this down- and upsampling

method in their paper about Convolutional Mesh Autoencoders (CoMA).

The paper by Chentanez et al. [9] that this work is mainly based on, focuses on the deformation of cloth and skin. They base their paper on the work of Gong et al. [22] and further adapt the spiral convolution, as well as the pooling. Furthermore, they propose a new architecture for their neural network. They apply their new methods to restore details from PCA-based clothing or create clothing from poses. Similarly, they construct the skin of hands based on the pose of a hand, which is especially relevant for this thesis. They improve on the spiral convolution by distributing relevant vertices of a spiral evenly around a center vertex. An equal distribution around the first ring around a vertex is used, as well as an ellipsis that is used to distribute more evenly in comparison to the preceding papers. For their pooling function, they implement a function that collapses edges based on a square error metric but apply rules that allow for an independent edge collapse. This allows parallelization of the edge collapses and a faster execution. During their unpooling process, they do not use barycentric coordinates but copy the value on an input vertex to the removed vertex, "akin to nearest neighbor upsampling" [9].

For their reconstruction of 3D meshes from poses, they do not directly infer the vertex positions in world space but predict the displacement of vertices for a less detailed mesh, which is created using a simple linear blend skinning (LBS) approach. Therefore, the CNN only needs to learn the difference between the low-frequency mesh generated through LBS and the high-frequency, detailed mesh generated from an offline simulator using a precise finite element method (FEM).

The idea of spiral convolutions has since been taken further by allowing for spirals of varying length or has been used in combinations for classification tasks of human activity [1, 7].

2.5 Mesh construction from poses

Spiral convolutions, as presented, are not the only way to construct meshes, or hands in particular. While the impact of the papers presented in this section on the contents of the thesis is little, they provide a window into research following different ideas.

Ge et al. [20] use a graph based CNN [12] to achieve the reconstruction of a 3D Hand. They create an end-to-end system, which generates the hand directly from a single 2D RGB image without depth cues. Their hand model is generated directly without using a predefined model to which deformations are applied. They first train with ground truth models and images created from them before switching to fine-tuning. They propose a novel approach that works on real-world images with weak supervision.

Moon, Shiratori, and Lee [45] also use a supervised approach to create a deep learning network that can create a mesh output based on a pose. Pemasiri et al. [51] propose an altered version of a graph convolution in combination with a GAN architecture to create a 3D mesh from a single image.

Choi, Moon, and Lee [10] creates meshes of humans and hands from 2D images with spectral graph convolutions. Chen, Wang, and Shum [8] uses neural radiance fields (NERFs) to create a 3D visualization of hands from video. It also proposes the MANO-HD model referenced in 2.1.1.

3 Method and Implementation

Navigating the path to a fast neural network that can construct a 3D mesh of a hand based on a pose and is useable in the Unreal Engine involves taking many decisions and solving various problems. This section of the thesis explains how I dissect this journey, which decisions I take, and why.

The structure follows an almost chronological order and starts with sourcing and creating training data. Two sets of training data, their creation and purpose, and how they were normalized are explained. Next, the foundation on which the neural network is built, SpiralNet++, is addressed. More technical aspects of the program provided by Gong et al. [22], mentioned in 2.4, are discussed further. The most significant part of the methodology focuses on implementing the changes proposed by Chentanez et al. [9]. The construction of the spiral kernel is detailed, and critical algorithms are highlighted. The pooling and unpooling function is another important aspect of the changes I implement and are discussed shortly after. The methodology for developing the neural network is concluded with the architecture description and the training procedure. It follows the implementation of all components regarding the Unreal Engine. The central portion of this chapter is the deployment of the neural network in the engine. Lastly, topics such as data loading, hand tracking, and visualizations are explained.
Before digging into the technical complexities of CNNs and kernel construction, an essential cornerstone of any machine-learning project has to be laid:

3.1 Training Data

For the use case of generating 3D models of hands from poses, a large dataset of 3D models with their related poses is needed. Two general methods can create and compose data for this task into a dataset.

As CNN is a supervised machine learning approach, labeled training data is needed. In this case, this data consists of the 3D mesh of a hand and the pose of the hand, e.g., in the form of joint angles. The first method is to use 3D meshes of a hand that is recorded and estimate the pose of the hand. This suffers from the problem that the pose detection of a 3D hand might deliver inaccurate results. The more considerable difficulty, however, is finding and annotating a dataset of thousands of hands with very similar positioning, scanning quality, and mesh topology, variables that influence learning behavior later [36]. Additionally, creating or adapting a dataset like that is a process of significant effort. Because of all those factors, which do not guarantee an overall gain, I do not consider such a dataset to be in the scope of this work and choose the second option: 3D meshes are created from generated or recorded poses. This allows the creation of meshes that are very similar from pose to pose. The pose has much less data, which is, therefore, easier to control for.

Chentanez et al. [9] also decided on that approach and used an offline finite element method (FEM) with a Neo-Hookean material model to generate their hand models. They used models with 33,000 vertices as the basis for their simulation. They used 5,000 animation frames and regressed the mesh from 18 joint angles.

3.1.1 NIMBLE

A variation of that approach is to use a different model to generate the hands from poses. Roßkamp used the NIMBLE project to generate hands with random poses and retrieve a 3D model [35]. NIMBLE is a “non-rigid hand model with bones and muscles”. It can reproduce realistic-looking hands (meshes and textures) based on poses. They use paramedic model learn-

ing to create the hand structure needed for the realistic rendering. Using a model like NIMBLE allows for the fast generation of a large training set that requires only a few further adjustments.

NIMBLE offers multiple ways to create 3D meshes. For this dataset, the generation from a random pose to an untextured mesh of the skin proves the most practical. The model NIMBLE creates has 5,990 vertices, similar to the faces of the CoMA dataset [52]. This will help the transition in SpiralNet++ when the NIMBLE dataset is implemented. A mesh that is too fine and has too many vertices might be harder to process for the neural network or require major adjustments to the model architecture. If the mesh is too coarse, it may fail to create a smooth and recognizable hand. The model is manifold, meaning it is without holes or borders, which allows for easier processing in the spiral convolution and is better for data handling overall.

Using those meshes, I noticed that many vertices do not connect to any other vertex. They do not belong to a face and do not refine the shape. Therefore, I wrote a small program using openmesh, a Python library for mesh handling, that identifies unconnected vertices and removes them [13]. This reduces the vertex count for each mesh to 4,994, almost 1,000 vertices less. Excluding vertices with no effect on the result improves the performance of the network because the information is more concise, and the network does not learn irrelevant patterns.

The pose data is available in two different versions in NIMBLE. First, the hand is represented using 3 degrees of freedom for every joint. This yields three angles per joint, four joints per finger, and five fingers for the hand. Overall, this pose is represented by an ordered array of size 20×3. Directly mapping angles to the hand allows for an intuitive understanding of how a pose works and which value influences which part of the pose. The other pose representation is a pose reduced by a PCA. It features only the 20 most important values for the pose but loses its real-world application and readability. With fewer values, the encoder of the neural network needs to compress the data less, which might be an advantage.

Nevertheless, I decided on the angle representation with 60 values. The human-readable format and the ordered joints were decisive in this decision. It is also beneficial for mapping other hand models, which will be critical for communication with the Leap Motion Controller. I expect a small, fast-forward neural network to be able to compress the values to the size of the decoder input and extract its semantics adequately. Furthermore, it allows for simple integration into other software, like hand-tracking software, without preprocessing the data. This will ease the implementation process for Unreal Engine.

I use 10,000 pairs of pose array and hand mesh for the first dataset. They are randomly generated using NIMBLE. Each mesh has 4,994 vertices and 9,984 faces. A neural network trained with this dataset has the problem that more expressive poses, like pointing at something or forming a fist, were harder to reconstruct than others. I attributed this to the dataset. As it is created using fully random poses, "medium" poses that look like a half-opened hand are overrepresented. For a "flat hand" pose, for example, all joints must be open to maximum extent in at least one direction. With fully random values, those edge cases are few, even though it is a common gesture for humans.

3.1.2 Leap Motion

In a second dataset, I tried to overcome the limitations of the first dataset by creating more expressive poses that function as parameters for the NIMBLE model.
I use the Leap Motion controller to obtain more realistic poses and track the hand in real-time. As the controller was already integrated into the Unreal project, the pose of the skeletal mesh

that is deformed by the Ultraleap plugin, can be read out, parsed into a list of strings, and saved as a text file.

During the training with the random NIMBLE dataset, 10,000 meshes proved to be a reasonable amount, from which the model was able to learn well. With a set frame rate of 60, I created and saved a pose every 10th frame, which took about 17 minutes. I started the saving without tracking to have hand poses that describe the default pose. When the tracking failed during the 17 minutes of recording time, the resulting files could be filtered out later. The poses are recorded in two sessions. I performed various common hand poses like pointing, a fist, the peace sign, flat hands (closed and spread fingers), and interacted with my other hand. Generally, all possible hand movements were performed in a way to minimize remaining in a pose for too long to avoid shifting the data in that direction.

As the next step, the data was cleaned from poses without tracking and fed into the NIMBLE hand model. The hand model of the Leap Motion and NIMBLE function is slightly different. While the basis of the model is similar, I created two matrices that would be multiplied and added element-wise to the recorded pose. That way, a linear mapping can adjust the differences between the two models for each joint and each axis. I determined those adjustments manually through two different methods:

1. Simple static poses: In this approach, I chose a simple input pose that could easily be posed manually in the Unreal Engine. In NIMBLE, I adjust the matrices so that NIMBLE creates a visually equivalent pose. For example, a pose where every input is zero creates a perfectly flat hand with closed fingers in the Unreal model. NIMBLE hands tend to be more relaxed and natural. This leads to slightly curled fingers, which are spread lightly. The mapping can reduce those differences.

2. Recorded poses: While this approach is similar to the first one, the poses compared are very different. Here, I record a pose using the Leap Motion Controller and match the pose in the complex environment of a natural pose. This is helpful for poses that contain multiple fingers, in which spacing and overlapping are substantial. Only when using real poses the functionality of the recognition and mapping of the pose can be proven.

The dataset is then created from the remapped poses, similar to the dataset generated with random values. Differences between the models are explained in more detail in 3.7.3. The final dataset contains hand meshes from poses that are commonly used. It offers more variation and extreme poses, especially fists and grabbing motions, which are almost absent in the first dataset.

This difference in variation can also be expressed quantitatively. I calculate the explained variance ratio on the poses using a PCA. The PCA is commonly used in machine learning for feature reduction, but I use the explained variance as a measurement of how much the variance in the dataset can be explained by the first principal components [2].
As the poses of the recorded dataset are more expressive, a broader range of joint angles is expected. In return, this leads to a higher variance in the data overall because the components need to account for this broader range. It is expected that the explained variance for the first components is, therefore, higher, as these components contain a greater portion of the pose variations.

The dataset of random poses would show a smaller explained variance in the first components of the PCA, as it is more evenly distributed. This can be caused by the data being more alike or that its complexity is spread over more dimensions and, therefore, harder to capture.
Indeed, the explained variance ratio for the recorded dataset is higher, especially in the first component, which can capture $\sim 64\%$ of the variance of the data. The dataset created from

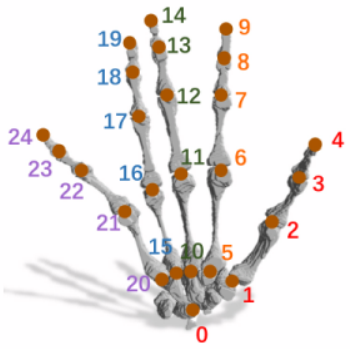


Figure 3: Order of joints in the pose representation of NIMBLE [35]

random poses can only gather $\sim 30\%$. In the first five components, this adds up to $\sim 93\%$ for the recorded dataset and $\sim 66\%$ for the random dataset.

The higher variance is also reflected during training: While it leads to a better model generalization, the main reason for this dataset is that it is also harder to train such a model. The same model might not capture the data's essence given the higher variance, and larger models and longer training may be needed [47].

3.1.3 Input Normalization

Normalizing the input data of a neural network offers many advantages regarding unit independence, regularization, and improved convergence [18]. In this case, Z-normalization is performed on the meshes. The data is shifted so that the mean equals zero and the standard deviation is one. A dataset D defined as $D = \{x_{i,n}, y_n \mid i \in f \text{ and } n \in \mathbb{N}\}$, with x being the input to be learned, the output y and the features $i \in f$. A sample of the dataset is specified as n. The normalization of the input follows as

$$x'_{i,n} = \frac{x_{i,n} - \mu_i}{\sigma_i} \tag{1}$$

as shown in [58].

The normalization is not specific to the sample but to the feature, in this case, for each vertex. Thereby, the position and deviation of the vertices to each other in a mesh are sustained through the normalization. The learnings of the autoencoder can be focused on the differences in the poses. It does not need to learn the general shape of a hand. The normalization process is reversible and can be applied after the neural network to receive functional vertices [22, 58, 67].

Normalizing the pose input, however, leads to significant performance decreases. For normalization and adapting the input poses, I tried the Z-normalization, min-max-scaling, where every value is shifted into the range between -1 and 1, as well as shifting the data in an entirely positive range, by adding π to each angle. All of those approaches led to worse average and median errors. Therefore, I did not normalize the poses for the training and inference of the pose encoder.

3.2 Adaptation from SpiralNet++

As a basis for this project, I use the implementation of SpiralNet++ provided by Gong et al. [22]. That paper is the basis of Chentanez et al. [9] as well and has already implemented an autoencoder with a fundamentally similar convolution approach. Chentanez et al. did not provide code alongside their paper.

Using an existing project allows me to focus on the specifics of this paper, the optimization of the neural network, and the integration with Unreal Engine and potentially other software. It also has the advantage of associating the procedures described in the paper and how they are executed in a project. Additionally, it decreases the required implementation time by a large margin since many core functions have already been implemented. The architecture of the project is also given but can be easily adjusted to the specifications of the new project.

SpiralNet++ project features multiple ways to use spiral convolutions and, therefore, offers multiple network setups. Interesting for this thesis is the reconstruction task, which features an autoencoder that should reconstruct faces from the CoMA dataset.
The spiral convolution layer and other important features of the project are organized independently of the autoencoder and the reconstruction task. This includes the spiral generation and the pooling function, which prepares the transformation matrices between the simplified meshes. The pooling function itself is not organized as a layer but as a simple function in the network.

While there are still many points at which the project requires changes to accommodate for the intricacies of this thesis, from data handling to the adjustment for network serialization (3.7.1), a solid base is given. The architecture of the project provided by Gong et al. [22] allows this thesis to focus more on the theoretical and algorithmic view and less on implementation. First and foremost, this is the spiral convolution.

3.3 Spiral Convolution

Convolution, in the context of deep learning, means the discrete two-dimensional convolution used in image processing in neural networks. In image processing, a kernel is convoluted with every pixel of the image. Convolution, as a mathematical operation, is a function over two functions. It is written with an $*$ and defined as such:

$$(x * w)(t) = \int x(a)w(t-a)da \tag{2}$$

In the context of neural networks, x represents the input, w stands for the kernel. The output of the convolution is known as feature map [23].

In image processing, this kernel is generally square, mostly 3×3, and holds the weights to convolute with. The surrounding pixels of each pixel are convoluted using the kernel. Here, they would be the input of the convolution operation. This process incorporates information about neighboring pixels and includes them within the pixel. This way, patterns and later complex shapes can be recognized.

Convolution on meshes can use a similar approach, where each vertex is a pixel. It is convoluted with a kernel of fixed size and that many neighbors. In contrast to images, the topology of a graph is far less direct. Whereas a 3×3 kernel of 9 pixels on an image works for most pixels, the neighbors in a mesh can vary in number. The distance to each neighbor can vary, as can their relative position in a 3D space.

3.3.1 Spiral Kernel

Spiral Convolution (SP) tries to overcome those challenges. Its goal is to define neighbors of a vertex so that a kernel of fixed size can convolute over them. An explanation for spiral convolution for a single vertex v_0 follows: SP uses a kernel of fixed size L. To find L vertices in proximity to the vertex in question, v_0, its surrounding vertices are sequenced in a spiral shape (4a). Gong et al. [22] defines the spiral as follows: The vertices around a vertex are ordered in k-rings and k-disks, where k is the number of rings. k-disks contain a k-ring and all vertices inside the ring. $\mathcal{N}(V)$ defines their neigbors, vertices adjacent to a vertex, for any vertex of the set of vertices V. The rings are defined iteratively:

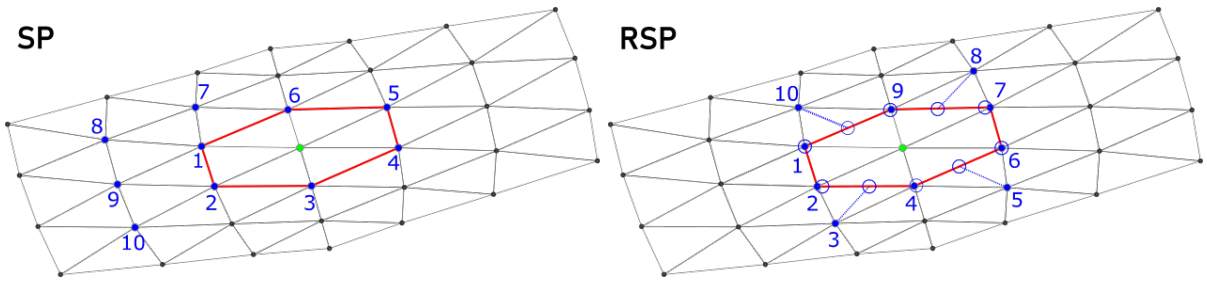
$$\begin{aligned} 0\text{-ring}(v) &= v_0 \\ k\text{-disk}(v) &= \cup_{i=0\dots k} i\text{-ring}(v) \\ (k+1)\text{-ring}(v) &= \mathcal{N}(k\text{-ring}(v)) \setminus k\text{-disk}(v) \end{aligned} \tag{3}$$

The 1-ring contains all vertices connected to the center vertex v_0. The 1-disk contains the first ring and the center vertex. The second ring then contains all vertices connected to all vertices of the first ring without the first disk.

The rings are concatenated until the number of elements exceeds the sequence length L. The sequence is then truncated to the length of L. The vertices in the rings are added to the kernel sequence until the sequence length is filled. If the current ring does not have enough vertices, the next outer ring is calculated. This way of generating a spiral only allows for two degrees of freedom: the rotation of the spiral and the starting position of the spiral. Gong et al. [22] decided on a counter-clockwise spiral rotation and an "arbitrary" starting point. While the spiral rotation should have no measurable effect on performance, a fixed starting position could have an effect. As the weights are shared for the convolution between the different vertices, a fixed starting point results in weights being convoluted with a vertex in a similar direction. An arbitrary position averages those effects.

Spiral convolution ensures that the environment of the vertex is captured meaningfully. The methodology is relatively simple and efficient to apply and only needs to be executed once at the beginning, provided that the topology of the meshes remains the same.

To include a larger area around the center of each vertex, Gong et al. [22, p. 3] also presents the "diluted spiral convolution". The diluted spiral convolution does not take every vertex of each ring until the sequence is filled, but every d^{th} vertex for a $d \in \mathbb{N}$. This way, farther vertices can also influence the convolution of the center vertex, and the receptive field is expanded. There is no added complexity in the convolution itself, as the sequence length L is not changed. This approach might be able to recognize larger patterns faster. This advantage is traded against the problem that closer, possibly more important, vertices are fully ignored. In addition, the runtime for the creation of the spirals could deteriorate slightly, as possibly more rings need to be calculated.



(a) Spiral Convolution approach (SP) (b) New Ring-based Spiral Convolution (RSP)

Figure 4: The figures ([9, p. 126]) show the previously used spiral convolution (SP) and the new ring-based spiral convolution (RSP). The spiral is defined for the center vertex (green). The blue colored vertices are part of the spiral sequence; the number specifies their position in the sequence. The 'one-ring' around the center vertex is marked in red. For the RSP, the sequence points are sampled uniformly on this ring, which are marked with blue circles. The sampled points are assigned to vertices in the vicinity. The assignment is chosen so that the sum of all distances of the assignment is minimal. Assignments are visualized with blue dashed lines.

An issue with this approach Chentanez et al. [9] sees in general, is that vertices in a mesh have a different number of neighbors. Therefore, a j^{th} vertex in a spiral can belong to the one ring of the center vertex, but on a different vertex, the j^{th} vertex may belong to the second ring. They, therefore, have a different position on the mesh and influence on the center vertex. Despite this discrepancy, both vertices will be accounted for with the same weight during the convolution.

Chentanez et al. [9] also see the potential that the spiral might be shifted to a side. L is often large enough to include all vertices of the first ring but not the second. Therefore, only a few vertices of the second ring can be included in the convolution and affect the center vertex. These vertices then lie on the same side of the vertex and shift the focus represented after the convolution to one side. As the starting point of the spiral is chosen arbitrarily and the number of neighbors for each vertex can be different, the direction of this shift can be arbitrary as well. Thus, it cannot be represented appropriately in the weights of the kernel. Chentanez et al. [9] argues that those challenges, which are grounded on the same problem, can negatively influence simulation results. The position of the shift is relevant, especially for physics-based applications, where the relative position of two points influences the forces between them.

They recognize the ability of SpiralNet++ to create working examples, but because of those difficulties, propose a new way of assigning vertices to the center vertex that avoids shifting behaviors.

To avoid the problem of shifting, Chentanez et al. [9] want to achieve an equal distribution of points around the center vertex. The idea is to create something closer to a circle or ellipsis than a spiral. This ring should then be able to include points of the rings around the center vertex and pick vertices from the next ring in a way that is equally spread around the center vertex. This way, a shift in the direction of the end of the spiral shall be avoided.
They propose two approaches to achieve this, of which I focus on the first: They determine the vertices of the first ring and sum up the length between each vertex to get the circumference of the ring. On this first ring, they distribute L new points at equal distances. The first point position equals the first vertex of the ring, and the next vertices are positioned on the ring at an equal distance to the previous one.

The points are equally distributed along the ring but do not represent a vertex of the mesh, as they are placed just by distance. In general, more points are distributed on the first ring

than the ring has vertices, so the remaining points that are not close enough to a vertex are distributed on the vertices of the other rings. For this, all vertices of the first three rings are used. To establish the relationship from each point back to a vertex of the mesh, Chentanez et al. [9] uses a rectangular optimization process. This optimization aims to minimize the sum of distances from a point to a vertex. A vertex must be assigned to each point; vertices must not be assigned twice. For this, there must be more vertices in the first three rings than points on the ring. This must be taken into account when choosing the sequence length. A problem like this is called an assignment problem [46].

3.3.2 Hungarian Method

The Hungarian Method is an optimization algorithm developed by Kuhn [33]. A variation of the algorithm is able to solve the assignment problem in polynomial time [15]. A common example of the assignment problem, less abstract than assigning vertices, goes as follows:

n workers shall finish n tasks in the shortest amount of time. Each worker is skilled differently and, therefore, needs a different amount of time to finish each task. The assignment problem asks which worker should be assigned to each task to finish all tasks the quickest. The time each worker needs is the cost. It can be visualized in a cost matrix, where each row represents a worker and each column represents a task. The Hungarian algorithm works on that cost matrix. The Hungarian algorithm follows multiple steps, here displayed as five:

The starting point for this algorithm is a square cost matrix. For the example of workers and jobs, the rows would specify a worker and each column a task at hand.

1. step: Subtract the smallest element of each row from all elements in that row. This makes the smallest element of each row a zero and eliminates negative values. In some cases, there is an obvious assignment which has a cost of zero. If that is the case, the algorithm can return.

2. step: If that is not the case, subtract the smallest element of each column from all elements in that column. This makes the smallest element of each column a zero.

3. step: Cross all zeros with a minimum number of straight lines. If the number of lines needed to cross all zeros equals the size of the matrix, one can go to the last step, “Assignment”. If the number of lines is less than the size of the matrix, continue.

4. step: From all entries that are not crossed by a previously placed line, pick the smallest value. Subtract it from all entries that are not crossed, then add it to all values crossed by two lines. Then, repeat the third step until its condition to proceed to step 5 is matched.

5. step: Assign each pair to each other, starting with a row with only one zero. That makes both unavailable, leading to another pair with only one free zero. Continue till the assignment is completed.

This algorithm is made for square cost matrices, where all assignments can be matched. It can be extended to rectangular matrices. Matches that should not occur can be avoided by setting a very high cost in the cost matrix [15].

For this purpose, a rectangular assignment is needed because the number of points in the first three rings of the center vertex is likely higher than the sequence length. Only by chance will the number of points be exactly the sequence length. If it is lower, the spiral convolution does not work as intended, and extra points have to be emulated to fill the sequence length.

The convolution operator of the neural network expects the same amount of points for every operation. Therefore, missing points in a spiral lead to a warning during execution of the network. Missing points are substituted using a k-d tree search that finds close-by vertices [59].

Runtime

The algorithm explained above has a runtime of $O(L^4)$ (L = sequence length) but can be optimized to $O(L^3)$ by choosing which numbers can be ignored first before proceeding to the next elements. This improvement of the Hungarian algorithm was independently developed by Tomizawa [61] and Edmonds and Karp [15]. In this use case, the runtime of the algorithm is not critical, as it only needs to be executed once at the start of the program. The assignments are saved and stay the same over the runtime of the program. Thanks to the similarity of the hand meshes, the spiral assignment only has to be done once for a single mesh, which is then applied in the same manner for all other meshes. Nevertheless, the assignment needs to be done for every single vertex of the mesh, and a quicker startup is always preferred.

3.3.3 Starting Point

Even though this approach is closer to an ellipsis than a spiral, it still has a beginning and, therefore, an order. In the implementation, the vertices belonging to a center vertex are represented as an array, which the convolution works with. Since the weights of the neural network are shared across the vertices, it is relevant that for each vertex, vertices with similar properties are in the same place. Gong et al. [22] chose to make the starting point "arbitrary" on purpose, but Chentanez et al. [9] oppose this sentiment. To make the position more similar, they proposed to always start with a vertex geodesically closest to a fixed point.
Choosing a direction instead of a fixed point can suffice for the same purpose. To keep the implementation close to the one of Chentanez et al. [9], I select a fixed point: the highest point in the middle of the bounding box of the mesh. The bounding box is calculated by taking the minimal and maximal positions of all vertices in each of the three directions. Assuming that the Y-axis points upwards, the fixed point can now be calculated as

$$(\frac{min_x + max_x}{2}, max_y, \frac{min_z + max_z}{2}) \tag{4}$$

The geodesic distance has to be calculated on unnormalized meshes, as the normalization changes the position of vertices and, therefore, their distances to each other. The distance to the fixed points is calculated and stored for each vertex in the mesh as an additional property. The true geodesic distance, which is way more complex than other distance metrics, is approximated by transforming the mesh into a graph, where the Euclidean distance equals the edge weight. Then, a Dijkstra path-finding algorithm can process the shortest way over the graph [14]. While this does not use the exact distance over the mesh, the distance over the topology of the mesh returns results of equal effect. When creating the spiral kernel, comparing vertex distances is easy and of constant time. When calculating the first ring around a vertex, the vertices of the ring can be compared. The list is then rolled over as many indices as the minimal-distance-vertex's index. This way, it is the first vertex in the list without changing the order of the vertices. When the points for the spiral are spaced onto the ring, the first point will match the first vertex. Thereby, the order of the vertices can be ensured.

3.4 Pooling

Pooling, in the context of neural networks, is an operation in which information is reduced by downsampling the input. The primary purpose of pooling layers in convolutional neural net-

works is to reduce computational complexity by omitting less important information. As image pooling is far easier to understand and the reasons and purposes for pooling data in a CNN are the same, I will use image processing as an example.

In a pooling operation on an image, four pixels are usually reduced to a single pixel. This is called a 2×2 Pooling. It is common for the remaining pixel to be either the average of the four pixels it replaces (Average Pooling) or the brightest of the four pixels (Max Pooling). Using a pooling like that, downsamples a picture by a factor of four. Processing a smaller image has the advantage that each kernel of the next convolutional layer can cover more of the original image. The convolutional layer, therefore, can abstract larger portions of the image. With each block of convolution and pooling, the visual representation diminishes while the semantic information increases [23].

Working on 3D meshes changes the way pooling works. As the vertices of a mesh are typically not in an Euclidean relationship, replacing every four vertices with a single vertex does not work reliably. A typical way to simplify meshes is by collapsing the edges of the mesh. The edge of a mesh is collapsed by creating a new vertex in the middle of the edge. Then, all edges adjacent to the two vertices of the edge to collapse are connected to the new vertex. Subsequently, the edge is removed along with the two vertices that form it. In the process of simplifying a mesh, in which many edges need to be collapsed, edges with the least relevance for the appearance of the mesh are determined and collapsed first.

Gong et al. [22] and Chentanez et al. [9] determine which edges should be collapsed by calculating the quadratic error for each edge. Edges with the lowest quadratic error are collapsed first. Gong et al. [22] determines the best edge to collapse iteratively and collapses that edge before finding the next edge until they reach the number of vertices they want to reduce the mesh by. They save their changes in a transformation matrix applied during training and inference. As I orient myself close to the implementation by Chentanez et al. [9], I implement their pooling and unpooling operations. They take pride in their novel approach to pooling that first determines which edges shall be collapsed and then parallely and independently collapses edges.

To make an independent collapse of edges possible, they maintain a priority queue of edges to collapse. For that, they use the quadratic error of the edge. When an edge is collapsed, edges adjacent to it are marked not to be allowed to collapse. This way, a multi-layered edge collapse and its dependencies can be avoided. Edges whose collapse would lead to a non-manifold mesh are also excluded. Chentanez et al. [9] formulate the terminating condition for the collapse of further edges as follows: Either no more edges can be collapsed, limited by the constraints above, or the quadratic error hits a threshold and the "number of vertices is less than 60% of the number of vertices at the start" [9, p. 126]. They further write, "The edges that are collapsed define our pooling operator." They do not describe how they apply their pooling during training and inference but reference the matrix transformation of Gong et al. [22] and Hanocka et al. [24] who do edge collapses, including determining the edges, during training and inference. As their paper gives no clear answer to this problem, and my implementation relies on a good performance during inference, I create a transformation matrix similar to Gong et al. [22] from the simplified mesh at each pooling stage. This is also based on the assumption that they would highlight the fact that their pooling would be fully done during network execution. Because I create a transformation list that is used for every mesh independent of its deformation, I do not implement true parallelization. In this approach, the pooling matrix needs to be calculated only once per training. The time to calculate that matrix can be disregarded compared to the overall training time. For the inference, the matrix will be baked into the serialized neural network and will not result in a performance loss.

Unpooling

Unpooling is an operation in which the input is upsampled. It is commonly used in autoencoders, where the output is the same size as the input. The unpooling operation should, therefore, match the pooling operation at the respective place in the autoencoder and produce an output the same size as the size of the input of the pooling operation. Assuming that an image was pooled with a factor of four, it also should be unpooled with a factor of four. For images, "Nearest Neighbor" unpooling is common. Here, three new pixels are created as neighbors of the original pixel, and its values are copied to the new pixels. This way, the image is visually unaltered but with higher resolution. The subsequent convolutional layer can then fill in information for each pixel.

Chentanez et al. [9] use an approach similar to nearest neighbor unpooling for 3D data. Vertices removed during an edge collapse are reintroduced by restoring them at the new center point of the edge collapse. This results in two vertices at exactly the same position, which will be separated by the next convolutional layer. Only vertices that belonged to an edge collapse during pooling are duplicated; other vertices are unchanged. This results in a vertex count equal to the mesh before the pooling operation. Previous publications like Gong et al. [22] and Bouritsas et al. [6] used barycentric coordinates to interpolate and restore vertices from lower-resolution meshes. For that, vertices omitted in the downsampling process "are mapped into the downsampled mesh surface using barycentric coordinates" [52, p. 6]. Based on those coordinates, the weights of the transformation matrix for upsampling a mesh are modified.

3.5 Network Architecture

The architecture of a neural network describes how the different components of the network are bound together and how they interact. Modern deep neural networks are built of layers of non-linear, differentiable functions that act on the input to form an output [23]. In this thesis, all parts of the CNN, convolutional layer, pooling layer, and activation function are seen as different layers. The combination of convolution, activation, and pooling layer, is called a 'block' in this thesis. This is mentioned because the literature sometimes calls such a 'block' a 'layer'. Together with the fully connected layer, those are the basic building 'layers' of the neural network. The core component is the convolutional layer that performs the main calculations.

The convolutional layer is paired with an activation function that allows the networks to produce non-linear expressions. Gong et al. [22] and Chentanez et al. [9] choose the ELU activation function, which works better and faster than the more common ReLU activation that is typically the default in machine learning [11, 23]. This, again, is paired with a pooling or unpooling function. A pooling function is used to decrease the size of the input. By decreasing the input dimensions, the following convolutional layers get access to a wider area of the image and allow the input to evolve in a more abstract form as the input traverses the net. The pooling operator reduces the mesh size by about 50%-60% per pooling. Those three layers form a block of convolution, activation, and pooling. Because of the new pooling and its non-parametric pooling factor, I follow the example of Chentanez et al. [9] and allow for multiple pooling layers within one block. This can change between blocks and allows more control over the pooling operation.

For the encoder, I tried variations of four or five blocks of layers following each other, reducing the literal information about the mesh but increasing the semantic one. The next layer is a fully connected layer. It connects all information in the convolutional layers extracted locally previously and combines it into densely packed information. The second half of the autoencoder is then set up in reverse: a fully connected layer followed by four or five layers of convolution

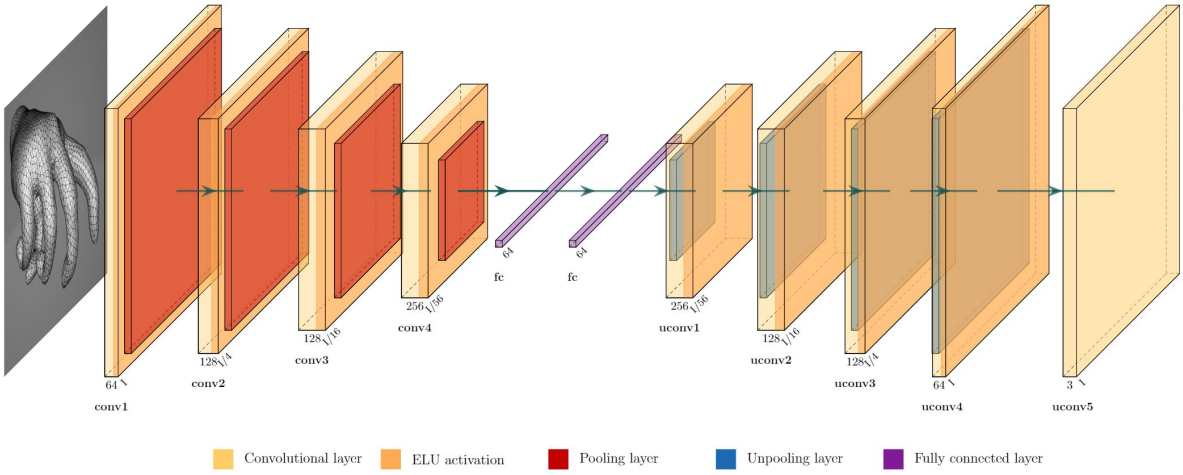


Figure 5: Architecture of a network with depth 4.

blocks with unpooling. Through this symmetry of the neural network, the output of the last block has the same channel size as the first convolution layer. This is generally much higher than the input channel size of three, representing a point's position. As the CNN needs a three-channel output to be able to output a mesh, a convolutional layer is attached, which reduces the number of channels back to three. When the network depth is mentioned, I am referring to the four or five blocks of convolution of one half of the autoencoder, even though the actual number of blocks and layers in the neural network is higher.

The network is trained to produce an output that matches the input, even though the information about the input is massively reduced to the middle of the neural network. A common feature used in CNNs are skip connections that connect the output of an encoder block in the input of the decoder block of the same size. The input of the decoder block is doubled in this process. It allows the decoder to better understand the input at that point in the encoding and is a tool to overcome the information bottleneck. They are used in Chentanez et al. [9] and can lead to significant performance improvements. Using the same decoder as before in combination with a different encoder prohibits the use of skip connection. The encoder cannot provide input to the skip connections, as it is no longer symmetric and does not hold the relevant information. Since the decoder is reused, skip connection can also not be applied in the first architecture, even though they might provide better results.

The second architecture is no longer symmetric as the encoder as the first half of the first autoencoder architecture is replaced. The encoder it is replaced with no longer encodes the 3D mesh, but the pose the hand holds. The pose comes from the NIMBLE hand model and contains way less data than the mesh before. Because of the small input data, fully connected layers can be used to build the encoder. The input size of the model as a whole, as well as the input size of the decoder, are to be considered in building the encoder. A basic structure of three fully connected layers and two ELU activations after the first and second layers is sufficient [11]. This new encoder can reduce the pose to a smaller semantic representation that fits the latent representation of the decoder. The pose encoder has changing layer sizes based on the size of the latent channel. The number of inputs should be increased or decreased steadily to match the latent space. The size of the input is 57, as the NIMBLE outputs the rotations of 20 bones in each direction. The first joint is the wrist, which is not moved and, therefore, can be omitted. $19 \times 3 = 57$ angle rotations remain. For a latent space of 32, the pose is reduced from 57 to 48 channels, then from 48 to 32. The last layer outputs the input size of the decoder, in this case

32. Therefore, no reduction, but a further abstraction is taking place.

For a latent space of 64, the channels are first increased to 60, then 64, and again 64 for the last layer. For a latent space of 128, the pose encoder needs to increase the channels substantially. From 57 parameters, it is increased to 64, then to 96, and finally to 128 channels. In those steps, the pose encoder needs to learn the latent representation of the previous autoencoder.

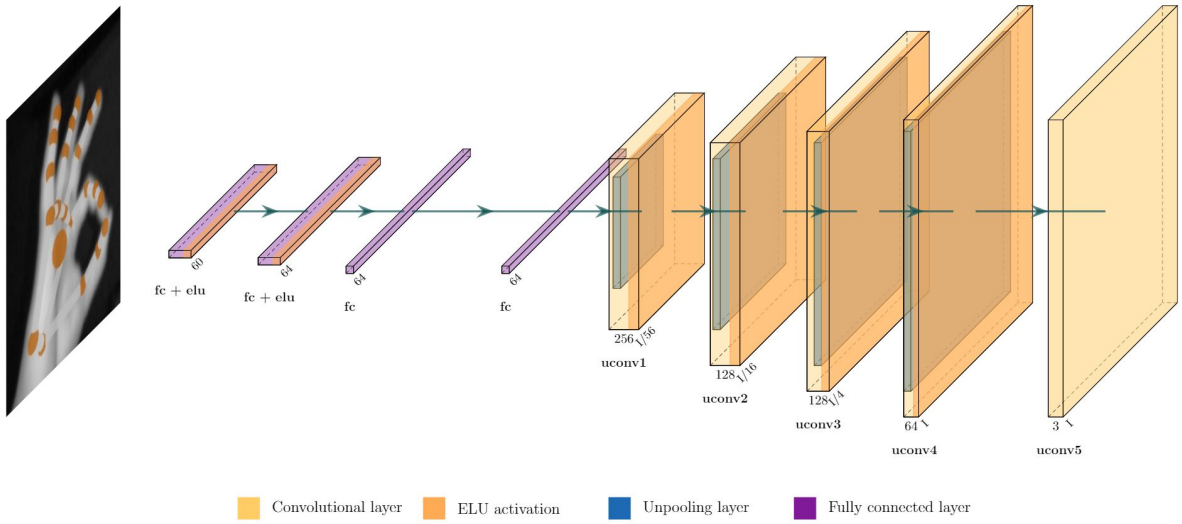


Figure 6: Architecture of a network using the pose encoder consisting of three fully connected layers. The input changes from mesh representation to the pose of a hand.

I use this structure for the CNN, in which I first train an autoencoder, instead of training the pose encoder with the decoder directly, for multiple reasons:
When training the autoencoder, the model can focus on learning a compact representation of the hand mesh. Only those features are relevant and allow the CNN to learn a low-dimensional latent space. Through this approach, the decoder learns how to reconstruct meshes and becomes a generator, which can create 3D hand meshes from the latent space. When the encoder is replaced with the pose encoder, the fast-forward network only needs to learn the small mapping between pose and latent space. In contrast, the mapping from the pose directly to the mesh is a highly complex task. The model needs to understand the geometry that is abstracted to the pose and craft a direct transformation between them. The model misses the crucial step of understanding the one part of the equation for which the latent acts like a bridge. Testing to train the pose encoder directly proved those problems again: The training showed very little to no training progress, which did not differ with different parameters. I expect the learning progress to increase once the model understands the basic mapping between pose and mesh. Nonetheless, splitting the learning task seems far more reasonable.

During the training of the pose encoder, the weights of the decoder are locked, so only the mapping between pose and latent space is learned, and the generating capabilities of the decoder are not changed negatively. Once the pose encoder is trained sufficiently, the weights of the decoder are unlocked, and the whole network is trained further. This is done to smooth out the differences between the pose encoder and the decoder and should increase performance slightly. For the analysis of the neural network, the architecture and parameters might change; the basis of the neural network, on the other hand, won't change.

To measure the performance of the network and thereby allow training it, I use the L1 loss, a common metric in machine learning. It represents the average vertex distance over all vertices

of the mesh. Given a vertex x_i, with i being the index of the vertex, x_i^g denotes the i^{th} vertex of a ground truth mesh. x_i^* denotes a vertex of the prediction of the neural network. The L1 error is then defined as

$$L_1 = mean\left(\frac{\|x_i^g - x_i^*\|_1}{3}\right) \tag{5}$$

in [9].
Chentanez et al. [9] also used an L2 loss term and a variation of the L1 loss that works on the normals of faces of the mesh. Because of their little influence in their work, the L1 loss seems sufficient for the approach in question.

3.6 Unreal Engine Integration

As soon as a well-functioning neural network has been trained, the issue of integration into the Unreal Engine arises. I present an overview of the solution I settled on before explaining other options and their benefits and disadvantages.

From the trained Python model, I create a modified version that expects a flattened input and reshapes it as the first step of the model. For simplified usage, I also include the normalization into the forward method. The model is then traced using the TorchScript 'Just In Time' (JIT) Compiler. This generates a single file that contains the trained network. This network can be loaded and executed by other Torch processes, which makes it very useful for integration into other systems. I use LibTorch, the C++ implementation of PyTorch, to load and execute the model. I create a simple Wrapper DLL that loads the LibTorch and CUDA dependencies [50]. This wrapper can then again be imported into an Unreal Engine plugin that controls access to the wrapper functions. Input and output are given as pointers, read, and filled out upon completion. The end of the inference call triggers an Event in the Unreal Engine, which other functions can subscribe to and retrieve the data.

To explain this way of integrating a PyTorch network in Unreal, I will first present an overview of three options. These options form the base to make neural networks in Unreal possible and shape most other parts of the integration before explaining implementations regarding the Unreal Engine.

3.7 Python Server

The first option is to run the model in an external Python process and communicate with it using a TCP server. The first significant advantage of this option is correctness. As the same Python code used for training is used for inference, it can be expected to behave the same. This is not necessarily true for other methods.

Another advantage is that the Python process can be run on a different PC and does not need to share the computing power with the Unreal Engine. This can improve performance and allow for a higher frame rate, as only the output of the neural network has to be considered. The network also profits from more available computing power, which could allow for a more extensive network architecture, which could lead to better results. Another advantage is that it is easy to implement. The Python code does not need to be modified, and similar publicly available implementations could be used to implement a server-client architecture. The main disadvantage of this method is the separation of the modules. Having the network as its own instance and process that needs to be shipped with the Unreal Project contradicts the idea of a complete system in which the functions of the CNN are available without a barrier. Additional

dependencies, like a Python installation, would be needed for this system. The promised performance benefits are only available when used with a second PC, which may not be possible in many applications. Otherwise, the execution of a network like that is expected to be slow, as it is not optimized for fast inference.

Even though later discussed performance benefits could also applied to a server-client structure, the level of integration does not match the targets I set for this paper. Therefore, I decided to pursue the other two options, which both utilize Neural Network Serialization.

3.7.1 Neural Network Serialization

In an effort to integrate the trained model into Unreal Engine, I use a Neural Network Serialization approach.
I originally planned to use the Open Neural Network Exchange format ONNX [49]. ONNX is already supported in Unreal Engine 5, even though it is mainly used internally for Mesh Deformation in an experimental plugin called 'Machine Learning Deformer' [19].

The original code of SpiralNet++ is unsuitable for serialization because the pooling operation uses sparse matrices. Only some functions are supported in the ONNX framework. Therefore, I rewrote the pooling function to use a different format rather than a sparse matrix, which achieves the same functionality with dense NumPy arrays. The new pooling function was tested and achieved equal results as the PyTorch implementation.

The exported ONNX file is supposed to run the trained network as if it would run in native Python. However, in multiple environments, the serialized and exported ONNX model resulted in highly different outputs than the PyTorch implementation. Meshes produced with the ONNX network suffer from huge irregularities, resulting in spikes on the surface. This was tested in UE5 and a minimal example in Python. The export of the ONNX model was also reduced to a minimal example and tested with multiple export options. The most promising option is to set a specific operation-set version to ensure all operations used in the exported ONNX file are supported. This problem occurred on Windows 10 and Ubuntu 22.04, with different CUDA versions and different PCs, and could not be resolved.

GitHub issues with this concern exist but have not led to a solution to the problem. On an issue I posted, a commenter recommended setting the model in train mode instead of evaluation mode because some layers, like batch normalization, can behave differently. Dropouts are also disabled when in evaluation mode. Those suggestions did not affect the ONNX export meaningfully and could not improve the final ONNX trace. Another suggestion from the same commenter was to trace a TorchScript trace with ONNX instead of the original model. The commenter did not provide background information for this approach. Yet, the reasoning seems intuitive: If ONNX cannot map the functions of the original PyTorch model, the TorchScript trace may have simplified an aspect that ONNX could not handle. A second trace may, therefore, be able to represent the original model.

Nevertheless, the problems with the ONNX trace remained, which is why the integration with ONNX could not be completed. While this problem is irritating, Jajal et al. [28] show that about four percent of conversions fail, even though a minimal amount is due to of significant inaccuracies in the final output. At the time of the comment on this issue, development on this part of the project was already completed.

The closest suitable alternative to ONNX is TorchScript. A TorchScript script is a serialized version of the model as a single file or executable that contains the trained network. The

PyTorch JIT compiler can trace a function and track operations on an example input. This way, the function can be optimized using just-in-time (JIT) compilation. Similar to ONNX, some operations and control flow cannot be encaptured using the trace function; problems with those operations were fixed beforehand. In contrast to ONNX, PyTorch's JIT tests models after exporting. Those tests achieved a similarity of more than 99% compared to the regular PyTorch implementation. The TorchScript was also able to show visually that the similarity is satisfactory. Therefore, I use a TorchScript of the model in the Unreal implementation.

3.7.2 LibTorch for Unreal Engine

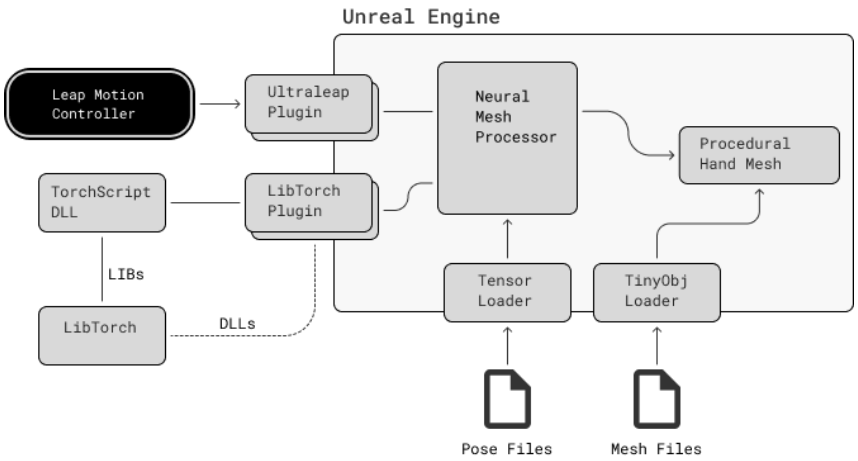


Figure 7: A simplified overview of the Unreal Engine implementation.

As the Unreal Engine only supports the execution of ONNX files but not the native integration of TorchScripts, an effective way of integration into the project is needed. For that, I use the C++ API of PyTorch: LibTorch. The LibTorch library provides many features of PyTorch, of which mostly basic Tensor operations and the execution of TorchScript models are needed. The Unreal Engine uses C++ for its implementation and projects; a C++-based library is, therefore, easier to integrate. To integrate the TorchScript model and the LibTorch library, I created a DLL that functions as a wrapper for the library and the execution of the neural network. It loads the libraries for LibTorch and CUDA for GPU support.

This new library exposes mainly three functions, which can then be called in Unreal: Two functions for loading and unloading the TorchScript and one forward method, which receives the input data and performs inference on the model. I then load this new library and the functions of the DLL in an Unreal Engine plugin called 'LibTorch Plugin', an adaptation of [64], thereby exposing them for the remaining Unreal project. A simplified overview of the structure is shown in figure 7. At the start of the plugin, the TorchScript is opened, and from there on, it waits for a forward call. Even though this way of integrating the neural network into an Unreal Engine project works and is fast enough for most use cases, it comes with some downsides. The installation process requires some technical knowledge because libraries for LibTorch and CUDA have to be installed manually and referenced in the build process of the custom library. Compiling a library for their hardware needs could pose a challenge to users with little technical background.

Another challenge faced was the unusual position of dependent DLLs in the Unreal Engine plugin folder. While not hard to overcome once discovered, this problem emerged with the switch from version 5.0 to 5.1 of the engine. Libraries would not be found if placed in the appropriate folder but have to be in the plugin's main folder.

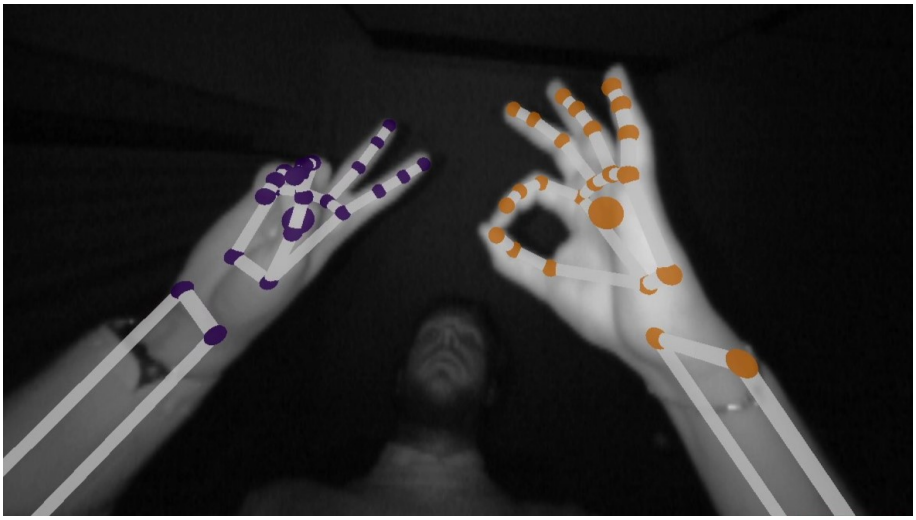


Figure 8: Leap Motion Controllers output input, overlaid with tracked points and bones, from the Ultraleap control panel. While overall tracking is good, notice how the thumb and index finger of the right hand touch, but the endpoints of the tracking visualization do not.

According to Sever and Öğüt [57] This approach is currently quicker than using ONNX in combination with the ONNX Runtime on the GPU, which would have been the alternative. While TorchScript and LibTorch work and create a precise, fast, and reliable integration, using ONNX is the preferred option and is easier for most.

3.7.3 Hand tracking

Integrating a hand-tracking system in the Unreal Engine is the proof of concept for the neural network and its integration into the engine. It allows one to visually compare the output of the network with one's own hand and proves that the neural network is fast enough to handle real-time applications. Testing specific poses becomes trivial, provided the hand tracking and the conversion between reference systems work well.

To ensure stable hand tracking, I use the Leap Motion Controller made by Ultraleap [63]. It is a small motion-sensing device that tracks and interprets hands and their movements in real-time. The Leap Motion is equipped with infrared LEDs and two near-infrared cameras that can read the reflected light of the hands in stereo. Those cameras enable it to track the position and rotation of bones and joints in a 3D space. The infrared cameras allow it to recognize bones through other parts of the hand, which makes reading poses that suffer from occlusion possible. The Leap Motion can be connected to a PC via USB. Their tracking software uses the raw output of the cameras to realize stable tracking and transform it into an easily readable format. This information is provided to other processes through their API. If placed on a table, its detection area covers a 3D space of about 60 centimeters, in which accurate tracking is possible. Those hardware features are supported by software that makes for an easy-to-use, robust tracking solution used in many fields [63]. It finds application in entertainment, healthcare, therapy and education, personnel training, industrial design, robotics, remote collaboration, or academia [53, 55, 63]. Everywhere where a touchless interface or the representation of hands is relevant, the Leap Motion can be an option.

Ultraleap, the company behind the Leap Motion Controller, provides, among others, a plugin for the Unreal Engine. This makes integration into the project simple: The driver and control panel for the Leap Motion need to be installed and configured to each need. The plugin needs to be

copied into the project's plugin folder and enabled. It comes with Unreal Engine Blueprints that already implement a connection to the Leap Motion API. The plugin includes skeletal meshes and Blueprints that map the reading of the Leap Motion to those meshes, resulting in meshes of hand poses. This way, a working hand-tracking solution can be realized very quickly.

To combine the real-time hand tracking of the Leap Motion with the neural network, the output of the Leap Motion has to be read and transformed into a format the neural model can process. The NIMBLE dataset was used to train the neural network; therefore, the Leap Motion data has to be brought to that format.

Through the skeletal mesh of the Leap Motion plugin, I can access 24 joint rotations that are relevant to the hand. In contrast to NIMBLE, those rotations are given in degrees instead of radians. Nimble only uses 20 joints, four less than the Leap Motion. The four extra joints can be attributed to the fingertips without the thumb. As they are not a real joint and are not used in NIMBLE, they were omitted. When reading the rotation and position of the Leap Motion tracked hand, the rotations are given in the reference space of the world. As I want to receive the rotation of the joint in reference from the previous bone to the next bone, the reference frame has to be reset. Therefore, I transform all rotations into the reference frame of the wrist joint. The rotation of the wrist is always 0° in all directions, the other rotations are then relative to the bones they describe. Another difference between the two systems is the direction of the angles. While a rotation towards the inner hand is represented in positive angles for Nimble, it is negative in the Leap Motion Output. A simple negation can equal this difference.

The starting point of both models is 0 degrees, with the joints straight, forming a flat, open hand. I implemented a linear function for each axis, to shift and adjust the rotational values, in which the negation could be applied as well. Another problem with the angles yielded from the Leap Motion readings is that angles during a finger curl motion could go over 180° and would then flip to -180°, resulting in wrongfully deformed hands. This was circumvented by using a dead zone for the rotation. Values over 180° are locked at 180°. Rotation lower than -90° are locked at -90°. This limits the possible "back curl" rotation of fingers to 90°, a rotation that should be limited by physical constraints of hands either way. The curling motion is limited at 180°, so joints cannot overlap themselves. Furthermore, the axes are swapped, which has to be considered but does not pose a problem. To improve the mapping between the two models, I use a linear model in which each joint angle is multiplied and offset with a specific value.

Finally, the conversion process is as follows:

1. remove four joints
2. transform to wrist space
3. conversion to radians
4. negate input
5. limit axis
6. swap axes
7. multiply input matrix
8. add input matrix

3.7.4 Data Loading and Processing

This chapter is about the input to the neural network that is not generated during execution but is predefined. One example of this is the training and test data. This chapter describes the loading and parsing of those inputs. This way of interacting with the Unreal environment and, therefore, the neural network is important for testing and analysis purposes but it could also be relevant if saves of real-time data should be played back. I implemented the loading of saved data for the autoencoder and the pose inference. Therefore, the Unreal project needs to be able to load 3D meshes and text files.

Meshes

As this thesis is not concerned with texturing, I settled on .obj wavefront files as the 3D input file format. While the features .obj files offer are limited, they are small, common, easy to understand, and human-readable. Even though Unreal offers the option to load 3D meshes from .obj files during runtime, I could not receive the vertices in the same order as in the original file. Since .obj defines its faces by the order of the vertices, the position in space of the vertices would be loaded correctly, but the faces could not form a surface. I, therefore, incorporated the TinyObjLoader library into the project [17]. This library contains a single file and allows loading .obj files from disks during the project's runtime. Provided with a file path, it loads vertices and faces in the correct order into the project. It does so quickly, in about 17 ms per mesh, which is suitable for this project, which may need to load one 3D object per frame. From then on, the loaded data can be passed directly to a procedural mesh or forwarded to the neural network.

Pose Data

Reading and parsing of the pose data is seamless and does not require external libraries. Poses are read from a simple text file in which angles are separated by commas. Three angles belong to a joint, separated as a new line. It makes the text file reminiscent of a CSV and very readable. To read it into the Unreal Engine, I use basic C++ functions to open and process a file. The described format is easy to parse and can then be forwarded to the neural network. Reading poses this way is very fast, about 0.4 ms per text file, and therefore suited for this processing.

Other file formats that contain more than one pose or mesh may be more suited for sequenced data, as the read process needs to be executed less often. The functions described can be easily modified for specific use cases.

3.7.5 Vertex Distance Shader

The performance of the neural network can be measured through error statistics and loss values, but a more intuitive way to convey performance is visually, especially for a topic in which visual performance matters. The reconstruction of hands falls into that category, which is why I integrate visual feedback into the Unreal Engine project. Creating visual feedback allows for conveying more information faster and more graspable, than in text form. It helps readers to understand the concepts better, and future research in this field may pick up points for improvement more easily. The visualization should be done in Unreal Engine, where all parts of this thesis converge. Users of the final project should be able to use the visualizations directly within Unreal without a third-party program or Python. The visualization is only possible for poses in the test or training data, or other poses where a ground truth exists. This excludes hand tracking (3.7.3), which is done through an inference process where the perfect outcome is unavailable. Visualization of the hand-tracking performance would need to process the NIMBLE simulation simultaneously, which cannot be done in real-time.

The static input data described above allows for comparing the neural model's output directly to the desired output or ground truth. When using the autoencoder, the output should equal the input. When using the pose as the input, the 3D model created with NIMBLE is available. Therefore, the corresponding ground truth can be loaded, and no time-intensive calculation is needed. With this data, a visual analysis can be used to better understand the neural model's performance and limits.

The vertex distance is a simple metric that shows the absolute distance between an original vertex and a vertex posited by the neural model. This distance is calculated for every vertex of the hand mesh. The maximum of this distance is determined and used as a reference frame. The reference maximum is reduced slightly, as parts of the metric's scale would not be used otherwise, as the maximum error is only reached in exceptional hand poses. The distances between original and generated vertex can be placed within this reference frame and mapped onto a color. This way, a heatmap on the surface of the hand is formed. It shows which areas the prediction is less accurate in compared to other areas.

Technically, this is realized by calculating the distances once the neural network yields its output. I use a three-way color linear interpolation for color generation, forming a heatmap between blue, green, and red. Once the distances are transformed into colors, they are set as vertex colors during the update of the procedural mesh. The material of the procedural mesh is set to a custom material that sets the vertex color of the underlying mesh as the base color. Another option is to set it as an emissive color, so discoloration due to lightning or shadows can be avoided.

4 Analysis

In this section, I showcase the performance and effectiveness of the model and the pipeline that it is used in. I compare the performance of the new RSP model based on Chentanez et al. [9] against the SpiralNet++ by Gong et al. [22] to see if I can replicate improvements in accuracy. Additionally, I perform a small parameter study to ensure the model performs best for my dataset and the use case. The autoencoder and the pose encoder undergo a quantitative and qualitative analysis. The latter shows hands created with this model in Unreal Engine with single color base materials and materials that visualize the margin of error.

4.1 Comparison with SpiralNet++

This thesis reimplements ideas of the paper by Chentanez et al. [9]. When implementing parts of a paper for which its source code was not published, it is interesting to see if their results can be recreated. However, a direct comparison between their work and this thesis is proving difficult.

Their work does not focus on the proposal of a new spiral operator but on the application of this operator. With the main application being cloth deformation, they created their dataset for cloth animation and compared the results of the spiral operators in that regard. The work on cloth deformation performed justifies the intricate set-up of their training and test data. This cannot be said about the recreation of said training data for a comparison between the models. Creating the cloth training data features upsampling and animation of upsampled cloths using physics simulation. Other aspects, which complicate a direct comparison with their implementation of the operator, are tied in with the training data. Because the network uses the low-frequency clothes as a predictor, their network only learns the difference to the high-frequency output, not the complete construction of the shape. Their autoencoder architecture features skip connections, allowing better information transfer between the encoder and decoder.

Because of those differences and the unavailability of the dataset, I perform a comparison with the SpiralNet++ model. As the difference in performance for multiple architectures and scenarios is well documented by Chentanez et al. [9], a comparison between SpiralNet++ and this implementation of their work is informative. It is expected that performance measures will slightly increase in contrast to SpiralNet++. The performance difference should be similar to the difference found by Chentanez et al. [9]. Nevertheless, Chentanez et al. [9] differences in performance could stem from better suitability for their dataset, their architecture, and the skip connections.

To create fair circumstances for both models and approaches, I compare my implementation of the ring-based spiral (RSP) to SpiralNet++. I use the parameters and architecture Gong et al. [22] describes, which enables not only a comparison with SpiralNet++ but also other networks compared earlier on the same network with similar settings and parameters. It remains that the performance of the new ring-operator can be expected to be similar.

Gong et al. [22] use the CoMA dataset, which consists of human faces with extreme expressions. With 5,023 vertices and 14,995 edges per mesh, the size of the data can be well compared to the dataset created with NIMBLE. The CoMA dataset consists of 20,465 objects, which are separated into 12 classes of facial expressions [52]. For this analysis, all objects and facial expressions are used to achieve a high variation in data.

For the architecture, I employ the architecture proposed for the reconstruction in an autoencoder in [22]. They use an encoder block made from a spiral convolutional layer with an ELU activation, followed by a pooling layer, that reduces the number of vertices by a factor of four. The autoencoder consists of three encoder blocks with 32 channels, followed by an encoder block

with 64 channels. A fully connected layer with 16 channels is added to "encode non linear mesh representations" [22, p. 7]. The decoder follows this pattern in reverse. As the channel size of the output of the last convolutional layer is the same as that of the first convolutional layer (32), a convolutional layer is appended to restore the channel size of 3, which fits the structure of the reconstructed mesh. Overall, the form of the neural network is very similar to figure 5, but with smaller channel sizes.

The differences in the implementation should not influence the architecture too much when comparing the different approaches to spiral generation and pooling. Especially the new pooling function creates the difference that only 50-60% of vertices can be removed. The number of vertices can not be specified precisely, and a higher simplification of the mesh is unachievable due to the limitation of not collapsing neighboring edges as shown in 3.4. Reducing the number of vertices by a factor of four can only be achieved by adding a pooling layer to each convolutional block. The pooling process is then executed two times directly after each other, thereby reducing vertices to about 28%. This leads to a fair comparison than using the same architecture, as the convolutional layers work on similar input sizes. That change in network structure increases the number of learnable parameters in the new variation. While the number of parameters for the SpiralNet++ is 154,899, the new pooling and spiral parameter uses 186,579 learnable parameters [22]. Even though this is an increase of about 20% in parameters, I consider it the fairest possible comparison.

An even more considerable increase in learnable weights would occur if the number of pooling layers would not be doubled, thereby decreasing the complexity of the mesh less. In increase in parameters would mainly from the fully connected layers at the center of the network, which are fully connected and thus grow faster in complexity with larger inputs. Since the input to the fully connected layer is about 16 times higher, if the pooling layer only uses a factor of two (in the best case), parameters would increase to 987'027.

Results

As expected, the test results of the different neural networks are very similar; mean and median differ only slightly. The new spiral operations and new pooling outperform the results of SpiralNet++, even if only slightly. The context of the higher parameter count remains. Despite the higher parameter count, the average time per epoch was reduced slightly (table 1). This results in a slightly faster learning time but is more significant for inference times when deployed, as the final software is meant to run in real-time. A visual impression of those results is shown in figure 9.

Model	Mean ± Std	Median	Time/Epoch	# Parameters
SpiralNet++	0.451 ± 0.576	0.248	24.55s	**154,899**
RSP	**0.421 ± 0.548**	**0.240**	**23.81s**	186,579

Table 1: The table compares the results of the reconstruction task on the CoMA dataset. The newly implemented RSP with new pooling achieves a slightly lower error for mean and median errors but also has more parameters. Nevertheless, the average epoch time is shorter than for the SpiralNet++.

Both networks learn well and consistently, with a steadily decreasing training error. The test error remains higher but close and predictable to the training error. This is true for both variations

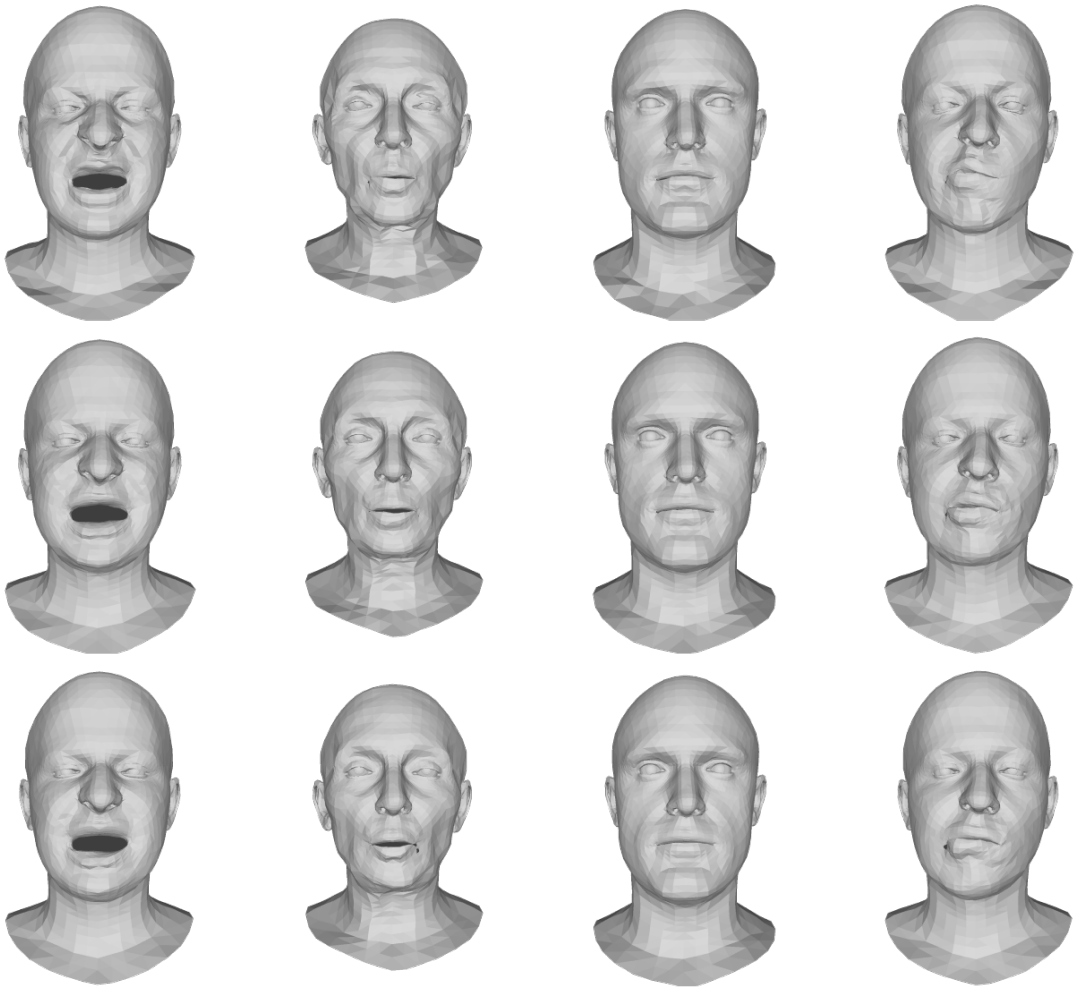


Figure 9: The image shows four example faces from the CoMA dataset from left to right [52]. They have different facial features and poses. For each pose, the ground truth (top row) and the reconstruction with the new ring-spiral model (middle row) and the SpiralNet++ model (bottom row) are shown.

of spiral networks. It shows that the training is effective and the models are not underfitting. The curves of train and test loss decrease steadily without overlapping, which signals that none of the models is overfitting either. Both models seem to converge after 300 epochs to a very stable, regular loss between epochs. Even after 200 epochs, the loss remains very close to the final result. The steep beginning of the graph signals fast learning, especially for the first epochs.

Based on what was learned from this first analysis, further studies on the model will use adjusted parameters for epoch and learning rate. As the models' loss only decreases minimally after 200 epochs, future training will be stopped after that many epochs. It reduces the training time by a third while still giving comparable results after that training time. It remains to be seen how this behavior will change as soon as the architecture or other hyperparameters change. Nevertheless, it enables faster testing of more variations. The learning rate seems quite high, and the loss decrease at the beginning of the training is steep. A smaller learning rate could improve results.

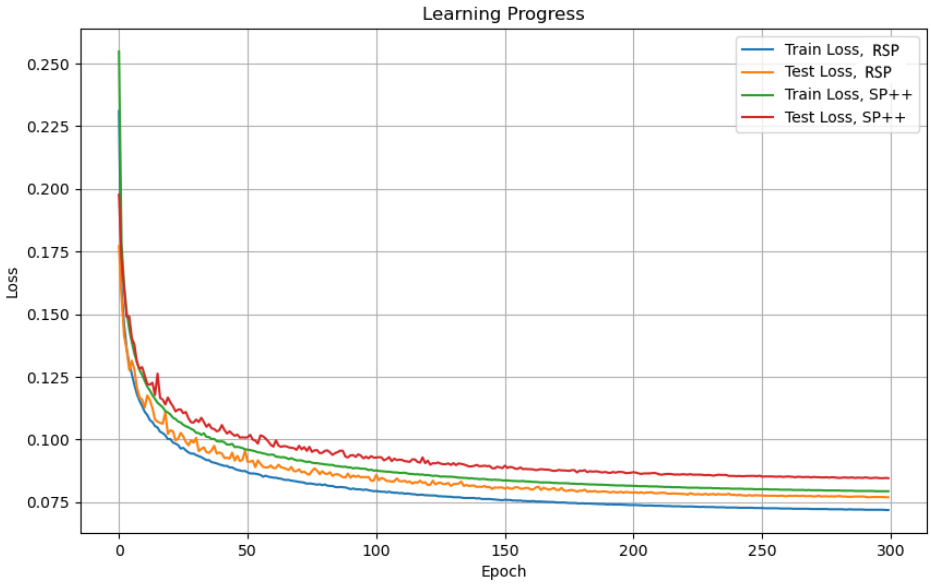


Figure 10: Loss curve for the new RSP and SpiralNet++ (SP++). RSP performs slightly better, but both curves show good learning behavior. Distances between train and test curves are stable. The curve starts to plateau after 200 epochs but does not seem to overfit after 300.

4.2 Parameter Study

Next, I perform a parameter study, that shall determine the best hyperparameters to achieve smaller errors and thereby increase its predictive probabilities. This also includes the goal of enhancing the generalization of the model. Different network architectures can improve model performance but must be balanced with other parameters, which can be explored in a study like that. A limiting factor of the network size is the planned real-time execution in Unreal Engine.

With the goal of a fluid hand motion during real-time simulation, reading the hand with the Leap Motion, extracting the pose, inferring on the network, adjusting the 3D mesh, and rendering must be concluded in less than 1/30-th of a second. For some simulation processes, like in virtual reality applications, higher frame rates of 90 FPS are desirable. Even though the final inference time in an Unreal environment cannot be tested in such a hyperparameter study, it can show how well a smaller architecture can perform.

4.2.1 Study Setup

This study was kindly performed on a PC in the CGVR workgroup, to which I do not have permanent access. The study was, therefore, designed with many preplanned configurations without the possibility of changing or expanding parameter options based on already tested parameter combinations. Therefore, when explaining the effect of each parameter I tested for, I also explained the parameter options I settled for. Values for the study are chosen from general knowledge about the training of neural networks and experience with the model from earlier trials, in which few parameter combinations were tested without a strict systematic approach.

The configuration of the study setup was discussed with my supervisor. The parameters selected led to a combination of 60 configurations that were tested. All parameter configurations were executed on the same PC, equipped with an RTX 4090 graphics card, under similar system conditions. All configurations were trained for 200 epochs, based on previous training experience

that showed comparable results to training with 300 epochs. The second dataset, consisting of 10,000 hand meshes created with poses recorded with the Leap Motion, was used for the study. Hyperparameters that were not modified are the batch size, which is set to 32, and the Adam optimizer with a learning rate decay of 0.99 [30]. I consider these values appropriate and do not expect any major change or improvement as a result of their adjustment.

For the study, I include these hyperparameters:

1. learning rate
2. spiral length
3. network depth
4. output channels
5. latent space

Learning rate

As mentioned in the previous chapter (4.1), the learning rate was quite high and learning was relatively fast. As, even with a few hyperparameters tested, the number of training runs increased fast, I wanted the network to train as fast as possible. As all architectures in this test will be more extensive with a higher parameter count than in the previous comparison on CoMA, I also include a lower learning rate. I settled for 3e−4, which seemed to work very well in preliminary tests. Even smaller learning rates, like 1e−4 were also considered, but for this study and the number of variations, training time should be as low as possible.

Spiral length

The length of the spirals is interesting as a hyperparameter. Gong et al. [22] used a length of 9 because it was determined by Bouritsas et al. [6], but not adapted to their use cases. Chentanez et al. [9] determined their best spiral length to be 13 and since then a paper with different spiral lengths in the network has been proposed by Babiloni et al. [1]. As the spiral in the convolution has the function of the kernel, and CNNs for image processing often use 3×3 kernels, a kernel of size 9 seems reasonable. Increasing the spiral increases the receptive field of the convolutional layer and could have a positive effect. Chentanez et al. [9] seem to have made that experience. To revisit their decision and explore the field into even larger spirals, spiral lengths of 9, 13, and 15 are tested. I expect the results to be slightly better with diminishing returns, especially compared to execution time with increasing spiral count.

Network depth

The depth of the network is a very relevant parameter of the network architecture. In this study, the network depth is given as four or five, which refers not to the network depth directly but to the number of encoder blocks in the encoder **??**. It is, therefore, directly correlated to the depth. On the one hand, a deeper network has more convolutional layers and more parameters, potentially producing outputs with lower errors. On the other hand, this leads to a longer processing time, which, in this context, is highly relevant. Additionally, in the configuration used in this study, an encoder block adds a pooling layer to the network. This effectively halves

the number of vertices after the last pooling. The most inner layer of the encoder is fully connected, and its input size is determined by multiplying the number of vertices by the number of output channels on the convolutional layer before it. This could also reduce the parameter count.

Output channels

'Output channels' refers to the number of output channels of the convolutional layer of each block in the network. The parameter is set as a list for the encoder blocks; the length of the list determines the depth as described above. With more channels in the network, more separate features can be extracted and accounted for. The bigger network potentially allows for lower errors in the final output. As I want to increase the network size compared to the architecture used for SpiralNet++, the following five configurations are used:

depth: 4				depth: 5				
64	64	64	128	64	64	64	128	128
64	128	128	256	64	128	128	256	256
				128	128	256	256	512

Latent space

The latent space is an the embedding of the semantic data into a multidimensional representation, to which the input data is compressed by the encoder. Its size specifies how much the data needs to be compressed [41]. It is, therefore, the size of the autoencoder bottleneck, making it a highly relevant parameter. This is especially of note as this use case does not allow for skip connections, like Chentanez et al. [9] use them. SpiralNet++ used a latent space of size 16. Since I expect better results with a larger latent space, 32 and 64 are used in the parameter study.

4.2.2 Results of the parameter analysis

The resulting error and the loss' development of the training were documented, and the last checkpoint was saved for possible future training or further tests.
When looking at the training data, it immediately became clear that many configurations did not learn properly. Figure 11 shows how this correlates to a too-high learning rate. The graph plots each training configuration as a colored point. Configurations colored in orange used a learning rate of 1e−3; green points stand for a learning rate of 3e−4. The graph plots the used sequence length on the X-axis and the median error on the Y-axis. Most of the training runs using the higher learning rate do not learn properly. Their loss suddenly increases to values over 0.7, while a typical loss for this training is lower than 0.08. Once an epoch produces such a high loss, the network cannot recover. An early stop condition aborts the training for many data points with very poor loss. This leads to very high final error values and a network that is unable to output anything meaningful. The sequence length, next to the learning rate, seems to have the most impact on that effect (figure 11). The number of parameters of a network does not seems to have an influence.

The data points of models that do not learn properly offer little insight into the other parameters tested. Therefore, they are removed from the next steps in this analysis.

The length of the spiral seems to have a much smaller effect on the data (figure 12). The best median error was achieved by a network using a spiral length of 9, closely followed by 13. The

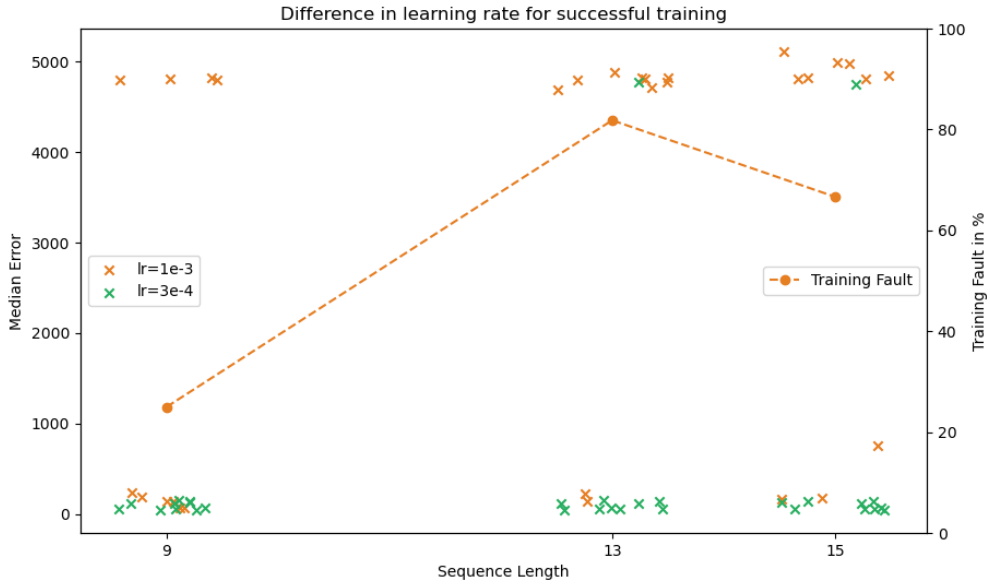


Figure 11: The graph plots the median error of training runs against the sequence length. The learning rate is given as color. Most trainings with a higher learning rate fail to learn properly, reaching median errors of over 4,000. The training fault visualizes the percentage of runs for a specific sequence length that failed. For a sequence length of 9, only about 22% of runs failed, while the failure rate was much higher for lengths of 13 and 15.

third-best result has a spiral length of 15. The differences, however, are minimal. It is noticeable that more training runs were removed from networks utilizing a sequence length of 13 and 15. The graph shows the data separated into two blocks of runs with close error values. The variance of the results is reduced with a higher sequence length, but the effect and the sample size are too small to make dependable statements about it.

The effect of the depth of the network seems to be similar to the sequence length (figure 13). Note that the depth is noted as described in 4.2.1. The best results are created by networks of depth four, with close contenders of depth five with similar low error. The dispersion along the Y-axis, showing the median error, is much higher compared to networks of depth five. The median error is plotted against the total learnable parameters of the model; as expected models with more layers have a higher parameter count and, in return, a lower error value. While the networks with the highest parameter count have a depth of five, they do not achieve the best results and can produce large errors. In that regard, networks of depth four deliver more predictable results. Those training runs form close clusters in the graph. What is interesting is that the models that provide the best results with a low parameter count all have a depth of five. This could be linked to the pooling, as the data is pooled once more compared to models of depth four. The extraction of information could be improved with further abstraction from vertices.

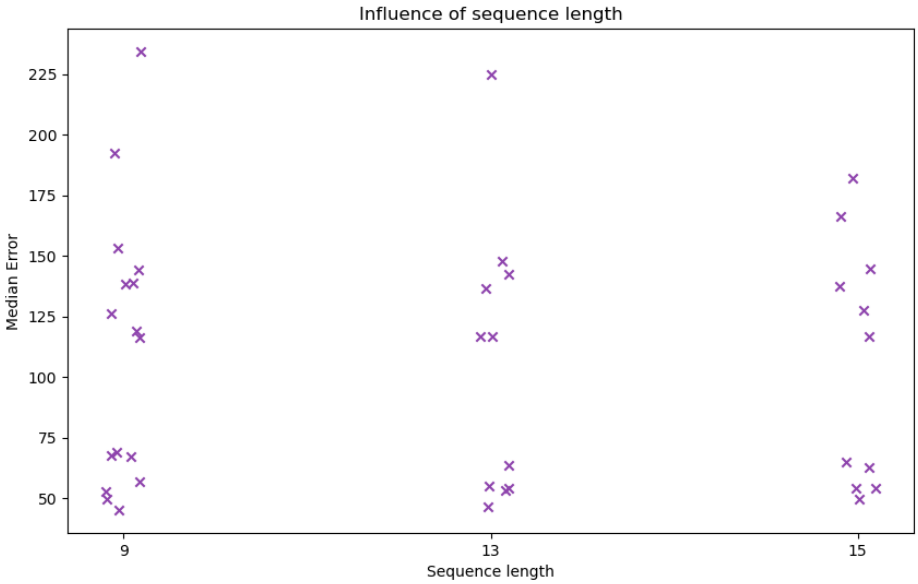


Figure 12: The graph shows the median error of successful runs compared to the length of the spiral. While mean error and variance decrease slightly with longer sequence length, a notable gap separates results from median errors lower than 75.

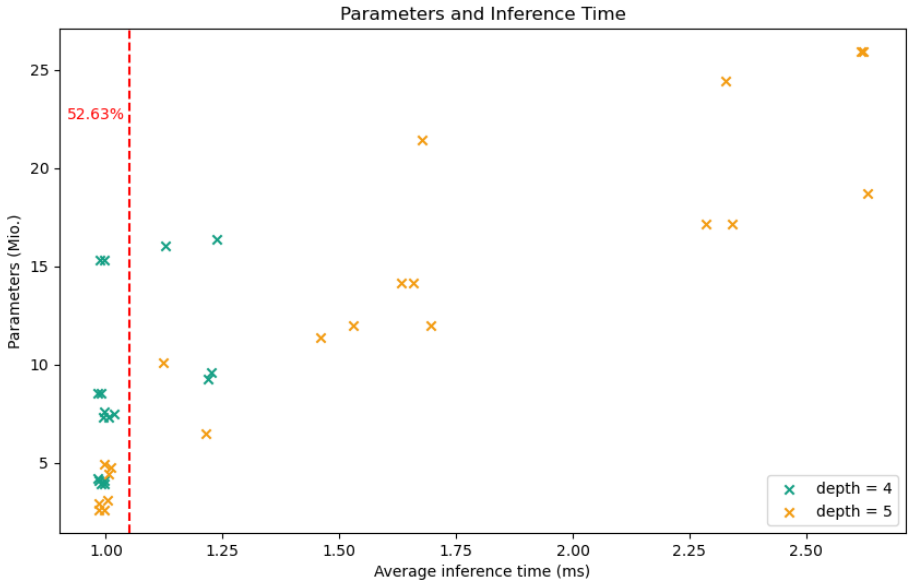


Figure 14: The graph shows the average inference time of a network compared to its parameter count. More than half of the tested networks can create results in about 1 ms. Networks of both depths and various parameter counts can compete in this area. However, networks with more parameters are substantially faster when only four layers deep. The slowest times are only represented by networks of depths five.

The execution time of the traced network, however, seems to be influenced by the network depth. For this visualization (figure 14), the number of parameters is plotted against the average inference time of the network in milliseconds. While about half of the tested networks generated

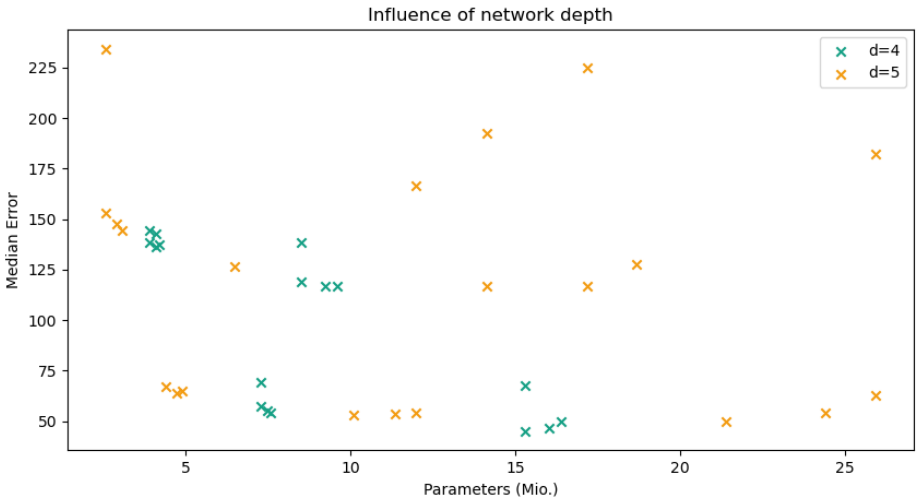


Figure 13: This graph examines differences in the depth of a network. While none of the test variations produce generally better outputs, deeper networks tend to have more parameters and higher median errors. The training runs form clusters of the same depth.

output in under 1.05 ms, there are clear trends that the depth of the network increases inference time. The depth seems even more significant than the number of parameters of the network, which also increases the inference time. Nonetheless, it has to be acknowledged that many networks with a depth of five execute in about 1 ms and have the lowest parameter count of the test.

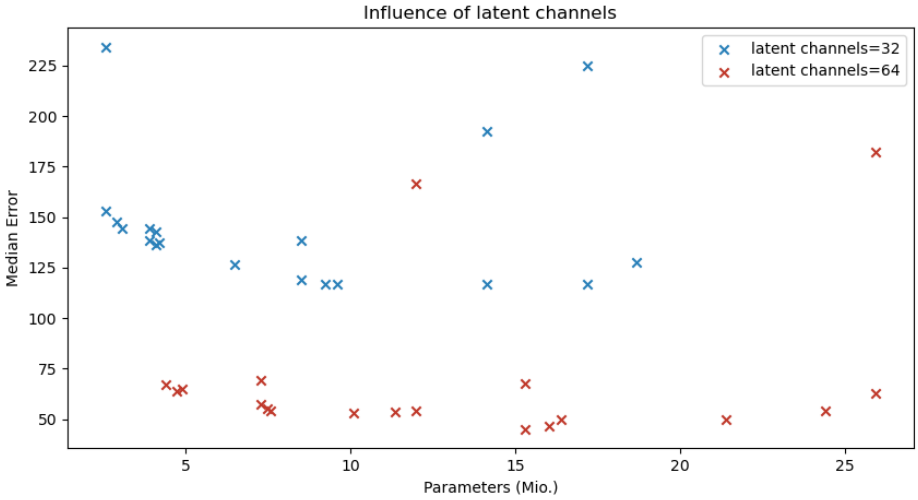


Figure 15: The training data in this graph is split into two well seperated classes by the latent space size: one with a latent space size of 32 and one of 64. The difference in performance is obvious, as almost all networks with a larger latent space perform drastically better. The size of the network parameter-wise has almost no impact in comparison.

With median error mapped against the number of parameters (figure 15), the grouping of datapoints shows the parameter responsible for the gap, seen in previous figures (13, 12): The size of the latent space separates the data almost perfectly along a horizontal line. The larger latent space of 64 produces a much lower error except for two occurrences. The bottleneck of the autoencoder, therefore, truly is the most relevant parameter. This is expected to some degree,

as the information about the 3D data needs to be less compressed, and more information about it can be forwarded to the decoder. Nevertheless, I expected a high parameter count or more output channels to be able to counteract it. The error is thus plotted against the parameter count and shows that its effect is marginal compared to the latent space (figure 15).

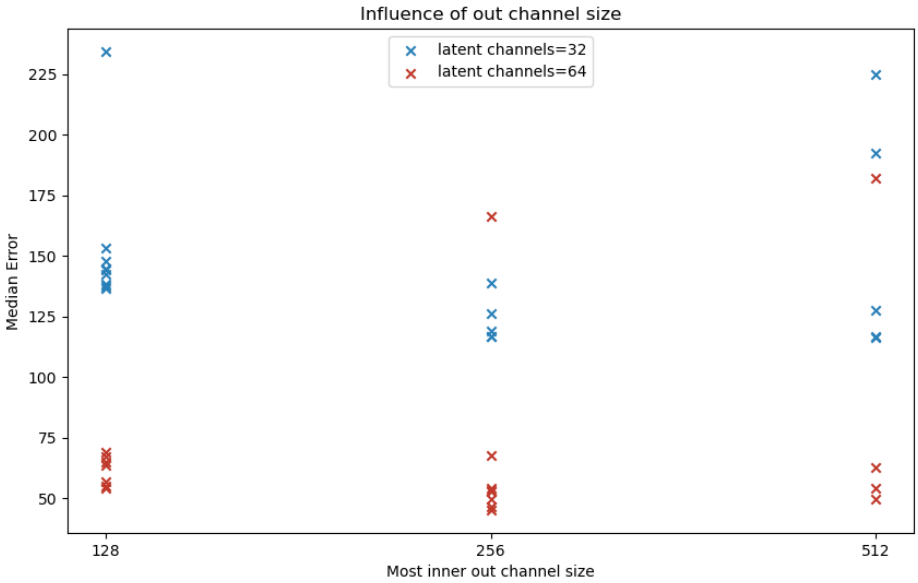


Figure 16: As the parameter size has not had a huge effect on network performance (figure 15) but an increased performance with a larger model is expected, the most output channels of a network are compared. The latent space again overshadows the difference in performance. More output channels increase the variance of the errors but has little effect on the median error.

To compare the channel size and, thereby, broadly, the network architecture, I compare the results by the number of output channels in the inner layer (figure 16). This can represent the architecture at large because all network architectures are staggered similarly but still allow the forming of groups from which observations can be pulled. Because of the significance of the latent space, I marked it by color in this graph as well. Architectures with maximum output channels of 265 perform the best in this analysis, closely followed by 512 and 128. The variance in errors between the network architecture is most prominent in this graph: Networks with a maximum of 128 channels per layer show minimal variance in error, except based on the latent channel. Results of networks with 256 and particularly 512 channels show greater scattering. Not reaching the best results and the greater scattering could signify less refined training. More extended training, a larger dataset, or a smaller learning rate could improve these results and benefit larger models.

The parameter study revealed the importance of the size of the latent space and its significant effect on the performance of the autoencoder models. Because of those massive performance increases, I trained four additional architectures with a latent space of size 128 for 300 epochs. The results of those tests confirm the expected trend: The mean and variance, as well as the median error, are reduced again significantly. The larger network architectures, however, lead to more parameters and longer training times. From figure 14, it can be concluded that this effect is likely to be noticed in inference times as well, with some parameter counts more than doubled, compared to the largest models of the previous test. Those tests can be seen in the context of prior training in figure 17.

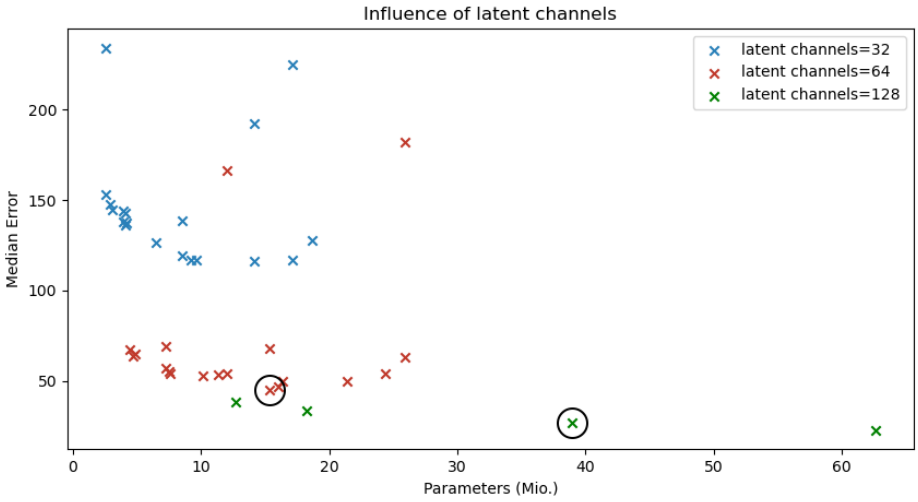


Figure 17: Tests with a 128 size latent space, integrated in the previous graph (figure 15). Notice the increase in parameters in comparison to previous models. The autoencoders used for the pose encoder are circled.

4.3 Pose Encoder

The training of the pose encoder is based on the principle, that the best-performing autoencoder is also the best starting point for training the pose encoder. The reasoning behind this approach is that the trained decoder only functions as a generator for a mesh. This should not influence the training of a pose encoder. It, therefore, makes sense to use the best autoencoders as a starting point.

I trained seven pose encoders in total: the best architecture proposed by Chentanez et al. [9], the best-performing autoencoder with a latent space size of 32 and 64, and all four models with a 128 latent space. All autoencoders were trained for 300 epochs to allow comparability, even though other parameters like learning rate and sequence length differ between them. The previously learned weights of the autoencoder are locked, so only the fully connected layers of the pose encoder learn. I select different layer sizes for the various sizes of latent channels because I want to use layer sizes that progress to the size of the latent channel. I use three layers for each pose encoder, as described in 3.5.

To train the pose encoders, I increase the learning rate substantially, as only the smaller pose encoder has to be trained, for which a learning rate of $1e{-}3$ is sufficient and allows for fast learning. The pose encoders were trained for 100 epochs. All other hyperparameters are defined through the underlying autoencoders.

As expected a very large model creates the best results, however, a slightly smaller model performs almost the same, with much fewer parameters and a shorter average training time. Due to its faster training time and the expected faster execution, I prefer the marginally worse model for further training. Again, with a slightly worse median error but surprisingly well performed, the best model with a 64 latent space. With a training time per epoch reduced by almost 50%, compared to the best model, I include it for testing and as an option for systems with less hardware capability or scenes with high rendering costs. The models with latent space 32 performed worse, as expected. The best architecture of Chentanez et al. [9] could not offer good

results but may have suffered from the large learning rate, as it can affect a smaller model more significantly. The pose encoders with an autoencoder as a base with more than one pooling layer per block performed worse than the other two models.

After training for 150 epochs, a large model with a latent channel of 128 stands out as the best model. It has a mean error of 1,149 and a median error of 983 but is very closely followed by a smaller network with a latent channel size of 64. It can achieve a mean error of 1,157 and a median error of 986. Both models have a high standard deviation: the larger model has a deviation of 767 around the mean, and the smaller model 791 (figure 18).

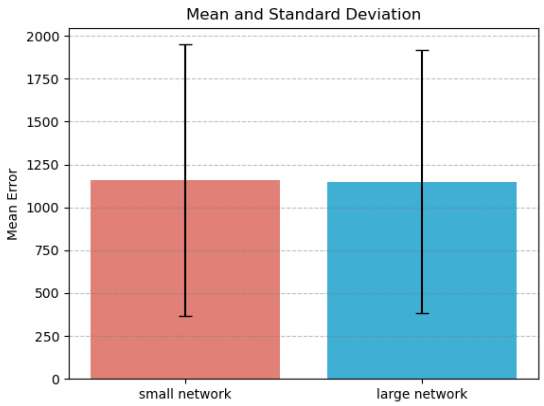


Figure 18: Mean and standard deviation of both networks with the pose encoder. Their performance is almost equally good, but overall error and deviation are relatively high.

Unlocking the weights to harmonize the encoder and decoder decreases the loss but does not positively affect the mean error. After 50 epochs of training the whole network, the mean and median error increased for both networks, which is a sign of overfitting. The training progress of both networks is shown in figure 19, with a particular focus on the training with unlocked weights. I, therefore, used the checkpoint of the pose encoder training after 150 epochs for further testing and execution.

The trace of the smaller model creates a 3D mesh from a hand in an average of 2.6 ms, and the large model in 4.2 ms. Both models allow for real-time application, but the difference in execution time is still very intriguing. The increase in average inference times comes from two categories of inference times. One of them is very fast with execution times under 1 ms. For the smaller network, about 90% of the time, it is that fast. Those differences are visualized in figure 20. For the larger network, it is less than 80%. Those slower executions take more than 10 ms, increasing the average disproportionally. The slow executions are even slower for the larger network, but the difference in distribution is crucial. In figure 21, the difference in that distribution can be inspected.

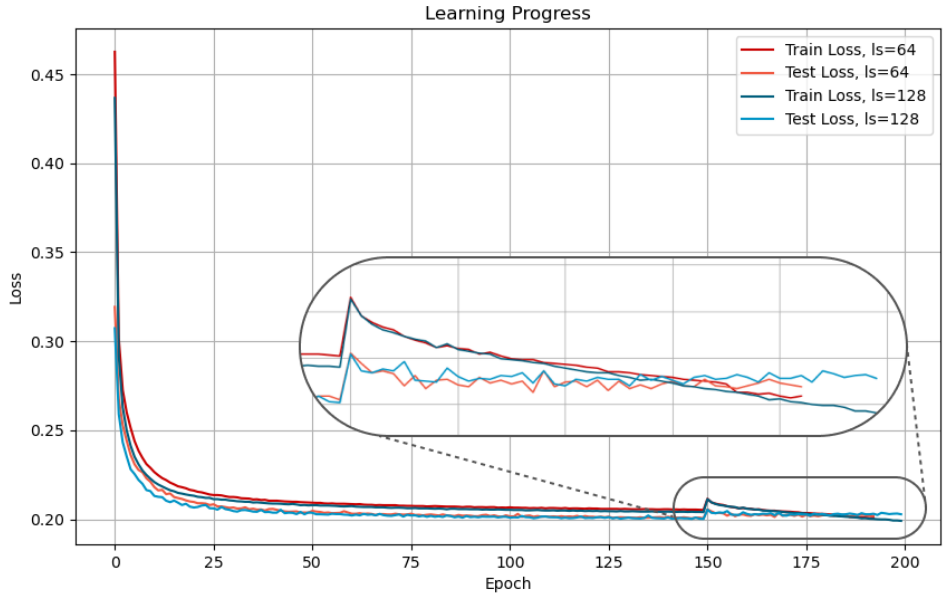
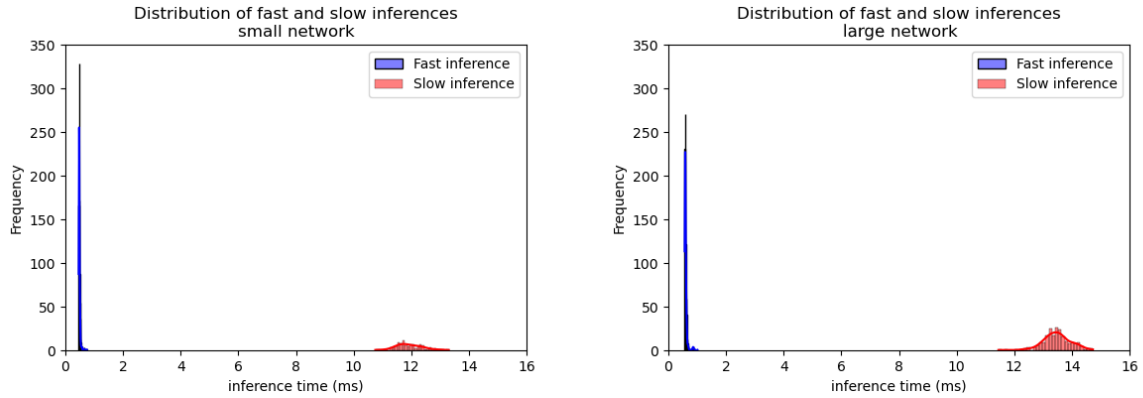


Figure 19: The loss graph shows the training progress of both pose encoders. The training is very steep but does not seem to overfit. Train and test loss are very close to each other over the training process. Training progress diminishes after 75 epochs. The weights of the decoder are unlocked after 150 epochs. Initially, it increases loss for both networks. The larger network is unable to benefit from unlocking the weights. While the test and train loss for the smaller model decreases further, their error does not, which I regard as a sign of overfitting.



(a) small network (b) large network

Figure 21: The histogram shows the distribution of execution speeds for the small and large neural networks for 1,000 runs. While the small network has slightly faster executions overall, the main difference in average execution times comes from the slow executions, which account for a greater part of executions with the large network.

Visually, the model is very close to the ground truth data. In most cases, the hand model is almost indistinguishable from the model created through NIMBLE. The general shape of the hand and the pose are translated well. Features of NIMBLE that simple linear blend skinning cannot accomplish, like the bulging of muscles under the skin, seem to be understood by the network and can be constructed from poses. There is no apparent difference between the models that use a size 64 and 128 latent space. For the visualization in this chapter, I therefore used the smaller model (figure 25).

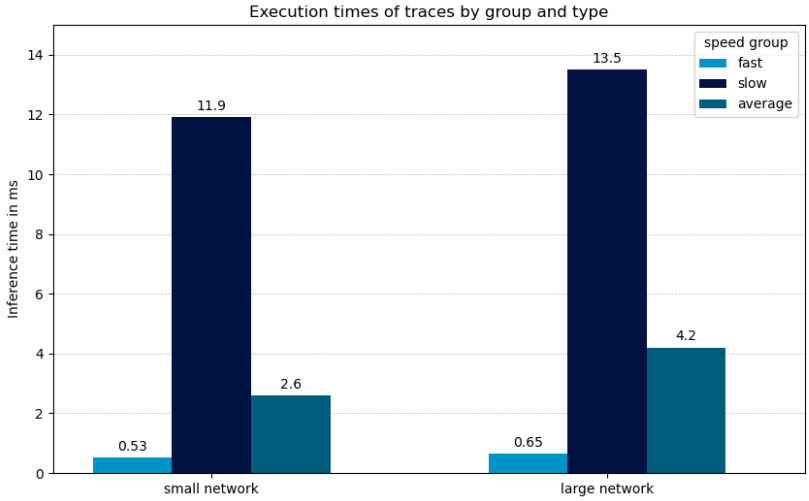
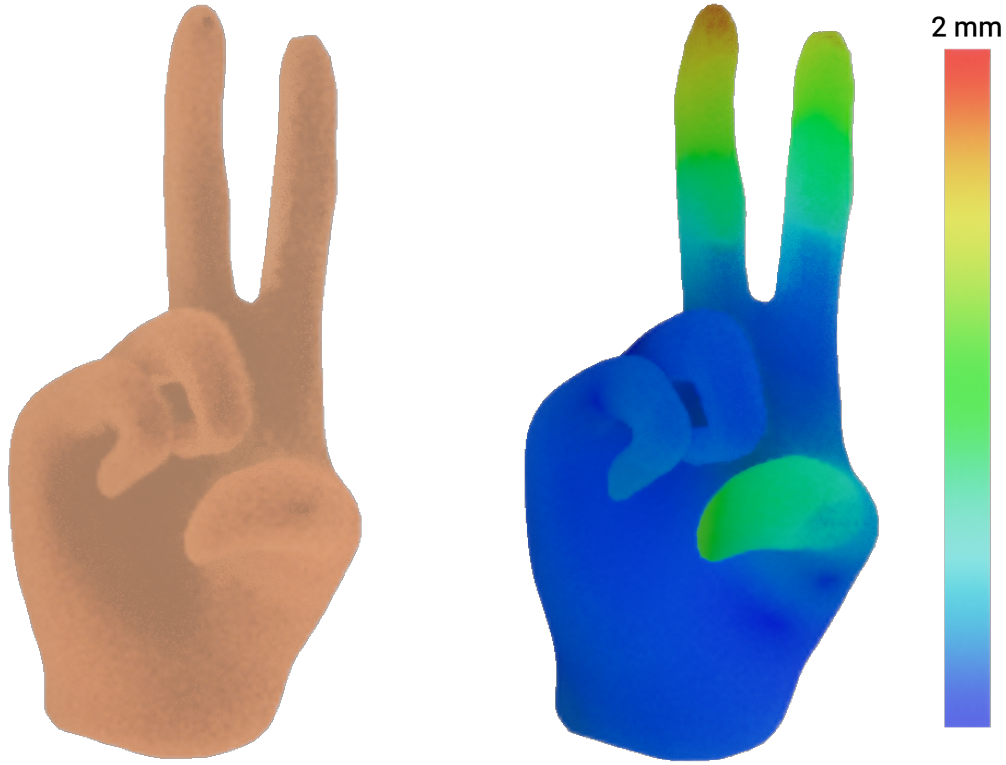


Figure 20: The graphic shows average times of inference for fast and slow executions, and the average over all executions, with times considered fast when execution is faster than 2 ms. Execution that take longer are 'slow'. The smaller network outperforms the larger network in fast and slow executions. Overall, the smaller network is about 38% faster. Times are separated, as the average times do not reflect execution time accurately, because of differences in distribution.

Overall, vertex distances increase from the wrist to the fingertips, with fingertips generally showing the most considerable vertex distances to the original mesh (figure 25). The top of the hand, as well as the palm, show the smallest errors. The distance to the center of the hand seems to have an influence, as closed fingers typically show minor errors, as can be seen in figure 23. The network has the largest difficulties constructing an equivalent mesh when fingers are crossed over and side-to-side movements in general. The thumb usually shows higher vertex distances than the other fingers, especially when moved under the hand in front of the palm. At no point, however, the neural network produces outputs that "glitch" or show extreme behavior that cannot be reasonably explained.



(a) Ground truth of a peace sign (b) Mesh generated from pose

Figure 22: A comparison between the ground truth of a hand and the mesh generated from its pose with RSP. The hand is showing the peace sign. The ground truth does not have a texture but a simple color. The mesh generated from the pose is colored according to its vertex distance, with a small vertex distance colored in blue and large vertex distances in red. Distances in between follow a heatmap shown next to the hand. Errors tend to increase to the fingertips and at the thumb, especially for extended fingers. The thumb is curved slightly downwards and does not touch the ring finger. As the pose is recorded with the Leap Motion, holding the peace sign in this position is difficult. Therefore, this is likely an effect of the imperfect mapping between Leap Motion and the NIMBLE model, as discussed in 2.1. Overall, the generated mesh is very close to the ground truth and the original pose can be easily understood.

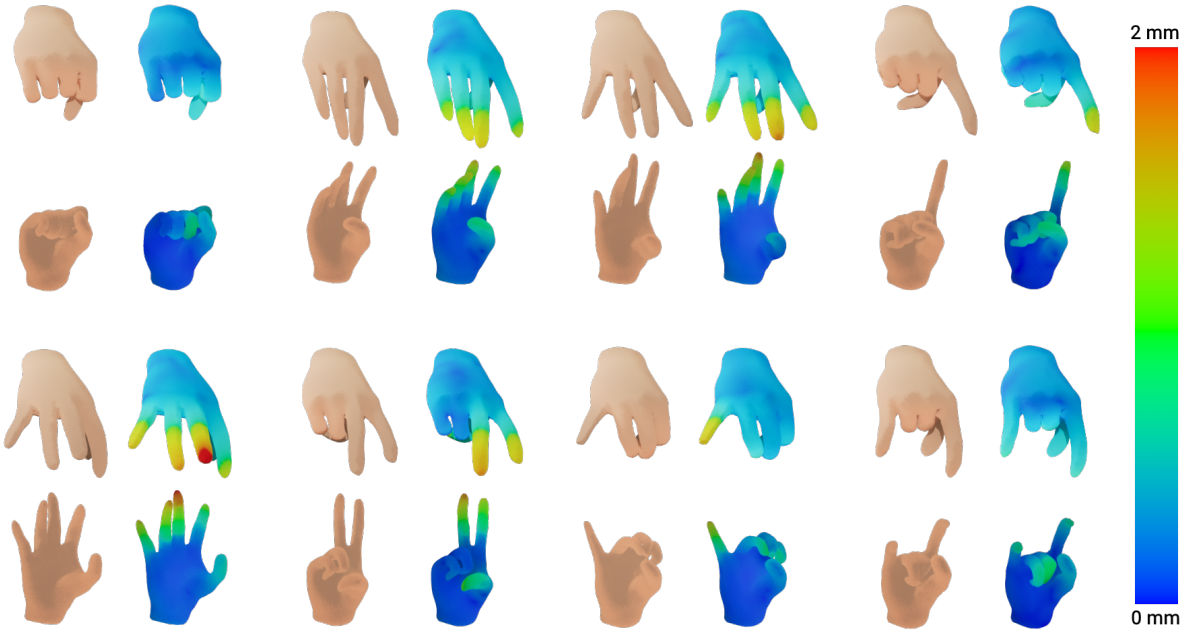


Figure 23: Comparison of eight poses from the test data of the LEAP dataset. Each example consists of two viewpoints that show the ground truth mesh and the generated mesh: a top view and a front view. The gradient of the generated mesh shows the vertex distance as described in figure 25.

When used in comparison mode with the ground truth data, one hand can be simulated at about 60 FPS. That includes loading the pose and the ground truth mesh, as well as the change of two procedural meshes. Executing the serialized model and copying input and output data to and from the library takes an average of $\sim 14\,\text{ms}$.

4.4 Tracking

Tracking is realized using the Leap Motion Controller as described in 3.7.3.
Tracking a single hand demonstrates better performance than the comparison with the ground truth when measured by the FPS of the editor. Because the neural model's execution time does not change, this increase must stem from the loading of data from the disk, which is relatively slow but not needed with the Leap Motion Controller.

At all times, using the neural model with the Leap Motion Controller, I felt that my hands would be represented by the output of the model. However, the error feels larger compared to the recorded test data. This certainly has to do with the imperfect tracking of the Leap Motion, but even with the direct comparison to its output, there sometimes is a feeling of disconnect. This is amplified by the curiosity to test the limits of the system. One can quickly iterate over poses and experiment with critical gestures. The greatest challenge with the network in a real-time setting is the positioning of the thumb, which leads to perceived disparities. The thumb tends to be either folded towards the hand, which works well for fists, or positioned outward from the hand. Aligning the thumb next to the hand without a gap proves to be a challenging task, even though the pose is common in real-world use.

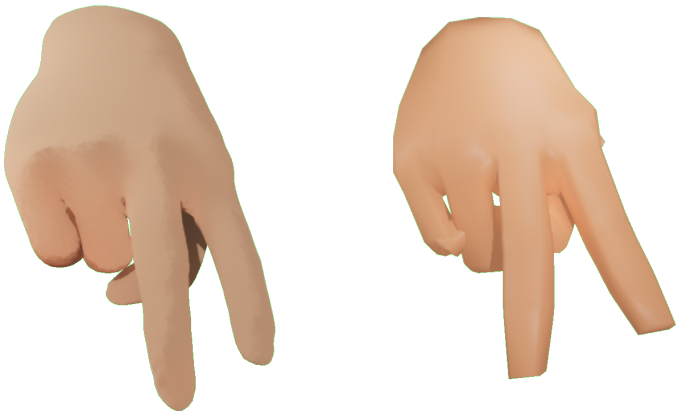
Additionally, an effect independent of the neural network is a ghosting artifact caused by the procedural mesh. Especially with swift movements, a shadow of the previous mesh persists in the scene long enough to be perceived. This is a problem of the procedural mesh and how it is rendered in the Unreal Engine. To my knowledge, the procedural mesh is the best, if not only,

way to render meshes that change in real-time, thus justifying no modification in the implementation of mesh visualization (figure 24).

Figure 24: The image shows the 'ghosting' artifact that fast and large movements can create in a procedural mesh in Unreal Engine 5. This extreme example was created using the Leap Motion Controller and shaking the tracked hand vigorously.

Nonetheless, I prefer the meshes produced by the neural network over the simple deformed mesh of the Leap Motion Controller plugin. Movements, in general, feel more organic and less stiff, and the muscles of the hand are shown and can be influenced.

Without performing a formal study, I asked friends and acquaintances to test the "hand simulation" without explaining the background of the thesis. Positive comments were made, especially about the fluidity, but also about the realism. Negative comments mostly regarded the discrepancies between the simulated hand and their real-world gestures. Those discrepancies were less criticized for the hand tracking of the Leap Motion Controller.



(a) Mesh generated from pose (left) and a skeletal mesh with the same pose (right), showing a peace sign

(b) Mesh generated from pose (left) and a skeletal mesh with the same pose (right), in a pointing pose

Figure 25: Comparison for two poses, the peace sign and a pointing hand, between the mesh generated using the RSP network and a skeletal mesh as it is used for the Leap Motion plugin by Ultraleap. The difference in output mesh is much stronger, and poses can not be recreated as perfectly as compared to the test data. Nevertheless, the mesh generated with the RSP model looks far more realistic, with tiny movements, like the deformations at the knuckles when making a fist (25a). The thumb can be a problem, but in the pointing pose (25b), the thumb wraps around the other fingers way more realistic, than the skeletal mesh.

5 Discussion

5.1 Comparison on CoMA

The motivation for comparing SpiralNet++ with the CoMA dataset was to measure the direct effect of the new spiral generation and the alternative pooling approach (4.1). For that, I chose an architecture that most closely resembles the one SpiralNet++ used in their reconstruction task. From the findings of Chentanez et al. [9], a generally improved convolution was expected. Chentanez et al. [9] pointed out flaws in the spiral generation and orientation that their approach would overcome. As I did not implement the pooling algorithm proposed by Chentanez et al. in a way that could utilize its parallelization potential, performance improvements could not be expected.

The training was executed for 300 epochs with a learning rate of 1e−3. From the analysis of the loss curve, which platoed soon after 200 epochs, I am confident in arguing that this training was not stopped too early. From the steep learning curve at the beginning of the graph, with the learning rate being relatively high for this application, I deem the learning process fast and well-tuned (figure 10).

After training for 300 epochs, the results show a slight performance increase in both mean and standard deviation and median error. However, those improvements can be attributed to the increased parameter count. Even when the parameter analysis on the RSP model shows that their influence is limited, it is fair to Gong et al. [22] to assume that they could achieve similar results using their model. Interestingly, the training time per epoch was reduced, which is also related to the inference time of the model. This faster learning time can most likely be attributed to the new pooling approach. Using max pooling and nearest neighbor unpooling eliminates a larger matrix multiplication as only an index select is needed for those pooling approaches. An increase in processing speed is measurable when accumulated over an epoch. A faster inference time of the RSP is especially relevant for use in real-time applications as planned for this model.

5.2 Parameter study

The results of the parameter study were discussed in chapter 4.2.2 sufficiently, as the results were relevant for the analysis of the final model, which processes poses as the input to form the mesh of the hand.

To summarize the discussion of results, it can be said that the influence of the latent space was much larger than anticipated. However, the parameter with the highest influence was the learning rate, which was chosen too high for many of the models, leading to them not learning at all. Other parameters seemed to have a dampened effect. Smaller and larger networks could achieve similar good networks, but the variance between the results of larger networks suggests that they could be trained for longer than 200 epochs or with more training data. When aiming for a network with as few parameters as possible, a deeper network with more pooling can deliver results similar to those of larger networks. This may be an effect of the fully connected layer at the center of the autoencoder, which needs fewer parameters when the input is reduced through pooling.

Overall, I consider the parameter study successful. With the size of the latent space, I found a specific parameter that could drastically improve the performance and determined parameters that work well for the autoencoder. The study also showed that smaller architectures can still reach good results, which can be important when using the model in environments where high performance is vital.

5.3 Pose Encoder

The final model, its integration into the Unreal Engine, and its execution in real-time are discussed in this chapter. It refers to the results in 4.3. The transfer learning of the pose encoder works well and is fast at mapping the pose to the semantic representation learned by the autoencoder. The training of the best-performing autoencoders showed that better performance is not necessarily transferred to the pose encoder. Error compared to execution times was especially impressive for the architecture with a latent space of size 64. The fine-tuning process, which aligns the new encoder with the decoder of the neural network, can reduce the loss and, in return, the error of the network further, but can also lead to overfitting quickly. The learning curves of all three learning stages suggest that the training process is sufficiently exhausted. More training, with parameters closer optimized to the use case and the exploration of even more extensive and more varied architectures, is certainly possible.Nonetheless, I consider the final architecture well-balanced based on the results of the processes I opted for.

The qualitative analysis shows only minimal differences. For the pose encoder, it is astonishing, especially when considering the speed of the model. While some poses work less well, like crossed fingers, a dataset focussing even more on varied poses could reduce those errors. Not only is the positioning of the fingers accurate, but the texture is smooth and only gets uneven in extreme poses. Errors increase outwards of the hand, which could be an effect of normalization, which multiplies outward values more than vertices close to the center of the hand. This also affects errors in the generated vertices. It is noticeable that the model does not generalize well, which is probably an effect of the training being very close to overfitting. Even though I did not use the retrained neural network with unlocked weights, a greater matching accuracy may have been achieved with an even larger dataset. Artifacts of the hand movements, like fingers clipping into the hand or each other, are rare and are also present in the ground truth model. Overall, the pose encoder works very well compared to the ground truth models and can simulate almost all poses of the test set. The absence of substantial errors in single meshes speaks in favor of a consistent model.

Hand tracking with the Leap Motion Controller and the construction of the respective hand mesh faces challenges (figure 4.4). The ghosting artifacts of the fast-changing procedural mesh, while sometimes distracting, are not so substantial that they would confuse or create strong reactions. I do not consider it that problematic, especially because it is only a small part of this thesis.
The overall tracking works well, and hand poses are presented accurately. The discrepancies between the real-world hand and the representation are noticeable, but from the results with the test data, they mostly seem to come from mapping issues with the Leap Motion Controller. The hand tracking feels fluid and does not stutter, which shows the viability of the approach. Small motions and the resulting skin deformations change the apperance of the generated hand in a way the skeletal mesh is not able to, as its mesh is strictly connected to its bones. Considering the current performance of hand tracking, it may not universally be the preferred solution, yet it remains a viable alternative for projects requiring a more realistic mesh deformation for hands.

5.4 Limitations

5.4.1 Reproduciblility

SpiralNet++ was the code foundation of this project, which made it easy to reproduce the result stated in Gong et al. [22] regarding mesh reconstruction, as shown in the CoMA comparison (4.1).

A significant part of this thesis is the implementation of algorithms presented by Chentanez et al. [9]. The description of the algorithms they use is sometimes vague or leaves questions unanswered. Since they neither provide their code nor formally explain their algorithms. From their description alone, I cannot guarantee that I followed their ideas as intended. This is especially true for the pooling, as the description is less precise and does not mention the precise algorithm for collapsing edges. Furthermore, I did not implement a parallelization of the pooling process.
Another limitation regarding the reproducibility of results made by Chentanez et al. [9] is the already mentioned difference in datasets. Their unpublished dataset is too elaborate to recreate and does not guarantee results that would be comparable. For those reasons, I view the reproducibility of Chentanez et al. [9] as a limitation of this thesis.

5.4.2 Computing Power

Training on my personal computer was slow and limited, not only by GPU power but also by memory and disk space limitations. This limited the experimentation with the neural network and testing of various architectures. While some training runs before the parameter study were possible, I would have liked more experience with the model and the training data in advance. Problems like the too-high learning rate, which invalidated many training runs of the parameter study, could have been avoided that way. While I developed on Windows 10 for compatibility with the Unreal Engine, the parameter study was performed on a Linux machine, leading to some problems and inconveniencing additional training runs. I would therefore like to thank the CGVR workgroup for supplying their hardware and especially my advisor, Janis Roßkamp, for setting up the program and collecting results.

5.4.3 Training data

The training data I use are recordings of my hand (3.1.2). The consequence of this is that it can only capture my hand structure and movements. The hand structure could influence that other people can perform the same pose, but their joint angles are recorded differently, with which the neural network is unfamiliar. Hand movements can differ even more severely. Some people can arch back their fingers way further than I can or move the ring finger and little finger independently from each other. However, those movements are not captured in the dataset I recorded, and the model I trained will have difficulty in accurately constructing meshes from those poses. More training data from different people could eliminate this problem and could diminish the effect in the trained model.

5.4.4 Mapping

As mentioned, another limitation is the mapping between the NIMBLE dataset and the Leap Motion hand model. This limitation was expected but could not be entirely avoided, and it is especially noticeable when using live construction with the Leap Motion Controller. The thumb, in particular, which was hard to map manually using a linear function, is less accurate than other fingers.
This limitation affects the actual usability of the trained model, e.g., in a VR application. The mismatch between the user's input and the model's output can differ and make the whole simulation feel less accurate. The mapping issues can also affect other areas of simulation, like hand

collisions. The model is not created or trained with collisions in mind, which one can notice quite easily. This regards deliberate collisions as well as accidental collisions.

The accidental collisions lead to overlaps of the fingers, either with each other or with the palm. Collision with the palm can occur when making a tight fist. In this case, a large part of the overlap is occluded by the fingers, yet it is noticeable. With complex finger movements, overlaps on the hands are also possible, even if less prominent.

By deliberate collisions, I regard movements in which fingers should touch each other or the palm. Examples of poses for this case are the flat hand with closed fingers or a fist with the thumb laying over the curled fingers. In those cases, the model is unable to bring fingers very close together. This behavior was already noticed when creating a mapping between the hand models, 3.7.3, but still occurs in the final model. Typical 'finger touch' poses can suffer from the same effect as in figure 26.

For better mapping of the thumb and other fingers, the mapping could be done using an optimizer or a more complex function that addresses the hand models of the Leap Motion and NIMBLE. An optimizer approach would also make the process less manual, eliminating human error. While this limitation influences the final output of the project, the mapping was not the focus of the thesis, and comparisons like in figure 23 show that the comparison with the ground truth is very close. Despite the limitations discussed here, I draw a positive summary.

Figure 26: Example for a pose in which thumb and index finger touch, but the RSP model (left) does not make contact. This case suggests insufficient mapping, which is especially difficult for the thumb.

6 Conclusion

A goal of this thesis is to implement the spiral algorithm described by Chentanez et al. [9] and thereby improve the quality of the spiral convolution algorithm in comparison to SpiralNet++. With this new spiral convolution approach, I want to create a neural hand model that allows for the fast construction of 3D meshes of hands. Target is the realism of the NIMBLE model used to generate training data. This neural model is to be integrated into the Unreal Engine, focusing on real-time capabilities that allow it to be used in hand tracking.

I implemented the better-performing variant of the spiral convolutions presented by Chentanez et al. [9] and compared it to the previous approach made by Gong et al. [22] (3.3, 4.1). While Chentanez et al. proposed improvements to the spiral generation, they could not live up to the expectations. The new spiral generation produces comparable output at faster training and inference times, thanks to a simplified pooling that focuses on max pooling and nearest neighbor unpooling. I thereby positively impacted the broader topic of deep learning on meshes.

The central theme of the thesis is the construction of hands using such deep learning models. For that, a dataset is made from realistic poses, which I recorded using the Leap Motion Controller to obtain realistic and common poses (3.1.2). These poses were mapped from the Leap hand model to a format closer to NIMBLE. From those poses, realistic hand models were created using the NIMBLE hand model, which uses a physics-based simulation based on an MRI dataset of hands.

I used this dataset and the new spiral- and pooling algorithm to train a neural network capable of producing 3D meshes of hands based on joint angles alone. The hand models it produces are visually close to meshes produced with NIMBLE (figure 23). I made sure to create a well-performing model regarding low regression errors and high inference speeds by conducting a hyperparameter study (4.2). It not only tests for typical hyperparameters like learning rate but, more importantly, tests domain-specific values like the spiral length and different network architectures.

I deploy a serialized version of this in Unreal Engine 5 using a customized LibTorch plugin (3.6). The handling of the neural network is combined with hand tracking software of the Leap Motion Controller and a procedural mesh generator, which allows for real-time hand construction from poses provided through the Leap Motion (3.7.3). The network size is adjusted to work with easily obtainable hardware and ensure real-time execution. The neural model is more than 1,000 times faster than hand construction with NIMBLE on the same hardware.

It shows that complex physics-based models for 3D objects can be simulated in real-time without previously simulated low-frequency input. This allows for relatively small, fast networks needed for resource-intensive applications like Virtual Reality. It proves, that this methodology can achieve visually accurate results even for domains as complex as hand simulation. Fine mesh deformation from underlying anatomy, as simulated in NIMBLE, can be learned in this approach while simultaneously creating large displacements like the fingers of the hand. An increase in realism for mesh construction has been shown, especially compared to the default mesh used by the Ultraleap plugin for the Unreal Engine 5. This improved realism can yield a greater immersion in virtual environments, a characteristic most VR experiences strive for. Success in this domain of hands with high complexity shows that the process applied in this study is usable and can also expand to other domains. Those can exceed hand simulation or VR to all environments in which fast deformation or simulation of 3D models is in demand. With the increasing integration of interfaces for executing machine learning models in various software and chip manufacturing that cater to those needs, approaches like this will become even more viable in the future.

6.1 Future Work

Scientific work based on this paper still has many opportunities to expand, beyond addressing the limitations stated above.

One way of expanding on this topic is by considering the shape of the hand. The NIMBLE hand model offers additional shape parameters, which represent hands of different shapes and sizes. This would allow for the estimation of a user's hand shape and its incorporation into the model.

The output of the NIMBLE model can, next to the skin mesh, also contain color and normal textures. While the color textures can greatly improve the visual resemblance of a hand, it can also help visualize the shape by naturally coloring creases and wrinkles. A normal map can help to detail the skin in ways the topology of the mesh does not allow for because of its coarse structure. If those textures could be simulated and projected on the procedural mesh, this could further improve the realism of the hands. This could also be achieved by using a CNN and learning to generate those images in a similar way to the approach in this thesis.

Interaction with objects is probably the most common use for our hands, yet this model does not acknowledge this. Hands deform differently when interacting with objects; for example, grabbing a box with sharp edges creates creases in the skin. The pressure applied to the hands at those points could be parameterized as input for the model. Integrating this aspect could bring another level of realism to virtual hands.

References

[1] Francesca Babiloni et al. "Adaptive Spiral Layers for Efficient 3D Representation Learning on Meshes." In: *International Conference on Computer Vision* (2023). DOI: 10.1109/ICCV51070.2023.01344.

[2] Ekaba Bisong. In: *Principal Component Analysis (PCA)*. Ottawa, Canada: Apress, Sept. 2019, pp. 319–324. ISBN: 978-1-4842-4469-2. URL: https://doi.org/10.1007/978-1-4842-4470-8_26.

[3] Federica Bogo et al. "Dynamic FAUST: Registering Human Bodies in Motion." In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). July 2017, pp. 5573–5582. DOI: 10.1109/CVPR.2017.591.

[4] James Booth et al. "Large Scale 3D Morphable Models." In: *International Journal of Computer Vision* 126.2 (Apr. 1, 2018), pp. 233–254. ISSN: 1573-1405. DOI: 10.1007/s11263-017-1009-7.

[5] Davide Boscaini et al. *Learning Shape Correspondence with Anisotropic Convolutional Neural Networks*. May 20, 2016. DOI: 10.48550/arXiv.1605.06437. arXiv: 1605.06437 [cs]. URL: http://arxiv.org/abs/1605.06437 (visited on 01/16/2024). preprint.

[6] Giorgos Bouritsas et al. "Neural 3D Morphable Models: Spiral Convolutional Networks for 3D Shape Representation Learning and Generation." In: *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. Oct. 2019, pp. 7212–7221. DOI: 10.1109/ICCV.2019.00731.

[7] Hamza Bouzid and Lahoucine Ballihi. *SpATr: MoCap 3D Human Action Recognition Based on Spiral Auto-encoder and Transformer Network*. 2024. DOI: https://doi.org/10.1016/j.cviu.2024.103974.

[8] Xingyu Chen, Baoyuan Wang, and Heung-Yeung Shum. "Hand Avatar: Free-Pose Hand Animation and Rendering From Monocular Video." In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2023, pp. 8683–8693. DOI: 10.1109/CVPR52729.2023.00839.

[9] Nuttapong Chentanez et al. "Cloth and Skin Deformation with a Triangle Mesh Based Convolutional Neural Network." In: *Computer Graphics Forum* 39.8 (2020), pp. 123–134. ISSN: 1467-8659. DOI: 10.1111/cgf.14107.

[10] Hongsuk Choi, Gyeongsik Moon, and Kyoung Mu Lee. "Pose2Mesh: Graph Convolutional Network for 3D Human Pose and Mesh Recovery from a 2D Human Pose." In: *Computer Vision – ECCV 2020*. Ed. by Andrea Vedaldi et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 769–787. ISBN: 978-3-030-58571-6. DOI: 10.1007/978-3-030-58571-6_45.

[11] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)*. Feb. 22, 2016. DOI: 10.48550/arXiv.1511.07289.

[12] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. *Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering*. 2016.

[13] RWTH-Aachen University Department of Computer Graphics and Multimedia. *OpenMesh*. URL: www.openmesh.org (visited on 02/22/2024).

[14] E. W. Dijkstra. "A Note on Two Problems in Connexion with Graphs." In: *Edsger Wybe Dijkstra* (Dec. 1, 1959). Ed. by Krzysztof R. Apt and Tony Hoare, pp. 287–290. DOI: 10.1145/3544585.3544600.

[15] Jack Edmonds and Richard M. Karp. "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems." In: *Journal of the ACM* 19.2 (Apr. 1, 1972), pp. 248–264. ISSN: 0004-5411. DOI: 10.1145/321694.321699.

[16] Roland Fischer. "Novel Algorithms and Methods for Immersive Telepresence and Collaborative VR." Bremen: Universität Bremen, CGVR, Aug. 18, 2023. URL: https://cgvr.cs.uni-bremen.de/publications/.

[17] Syoyo Fujita and et. al. *Tinyobjloader/Tinyobjloader*. tinyobjloader, Feb. 25, 2024. URL: https://github.com/tinyobjloader/tinyobjloader (visited on 02/27/2024).

[18] Keinosuke Fukunaga. "Chapter 1 - INTRODUCTION." In: *Introduction to Statistical Pattern Recognition (Second Edition)*. Boston: Academic Press, Jan. 1, 1990, pp. 1–10. ISBN: 978-0-08-047865-4. DOI: 10.1016/B978-0-08-047865-4.50007-7.

[19] Epic Games. *Using the Machine Learning Deformer*. UE5 Documentation. URL: https://docs.unrealengine.com/5.0/en-US/using-the-machine-learning-deformer-in-unreal-engine/ (visited on 02/27/2024).

[20] Liuhao Ge et al. "3D Hand Shape and Pose Estimation From a Single RGB Image." In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2019, pp. 10833–10842. URL: https://openaccess.thecvf.com/content_CVPR_2019/html/Ge_3D_Hand_Shape_and_Pose_Estimation_From_a_Single_RGB_CVPR_2019_paper.html (visited on 11/20/2023).

[21] Susan Goldin-Meadow. "The Role of Gesture in Communication and Thinking." In: *Trends in Cognitive Sciences* 3.11 (Nov. 1, 1999), pp. 419–429. ISSN: 1364-6613, 1879-307X. DOI: 10.1016/S1364-6613(99)01397-2.

[22] Shunwang Gong et al. *SpiralNet++: A Fast and Highly Efficient Mesh Convolution Operator*. Oct. 2019. DOI: 10.1109/ICCVW.2019.00509.

[23] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. 800 pp. URL: https://www.deeplearningbook.org.

[24] Rana Hanocka et al. "MeshCNN: A Network with an Edge." In: *ACM Transactions on Graphics* 38.4 (Aug. 31, 2019), pp. 1–12. ISSN: 0730-0301, 1557-7368. DOI: 10.1145/3306346.3322959.

[25] Shawn Hershey et al. "CNN Architectures for Large-Scale Audio Classification." In: *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Mar. 2017, pp. 131–135. DOI: 10.1109/ICASSP.2017.7952132.

[26] Anne Hollister et al. "The Axes of Rotation of the Thumb Carpometacarpal Joint." In: *Journal of Orthopaedic Research* 10.3 (1992), pp. 454–460. ISSN: 1554-527X. DOI: 10.1002/jor.1100100319.

[27] Xuan Huang et al. *3D Visibility-aware Generalizable Neural Radiance Fields for Interacting Hands*. Jan. 1, 2024. DOI: 10.48550/arXiv.2401.00979. arXiv: 2401.00979 [cs]. preprint.

[28] Purvish Jajal et al. *Analysis of Failures and Risks in Deep Learning Model Converters: A Case Study in the ONNX Ecosystem*. Mar. 30, 2023. DOI: 10.48550/arXiv.2303.17708. arXiv: 2303.17708 [cs]. preprint.

[29] Baris Kayalibay, Grady Jensen, and Patrick van der Smagt. *CNN-based Segmentation of Medical Imaging Data*. July 25, 2017. DOI: 10.48550/arXiv.1701.03056.

[30] Diederik Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization." In: *International Conference on Learning Representations* (Dec. 22, 2014).

[31] Ilya Kostrikov et al. "Surface Networks." In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2018, pp. 2540–2548. URL: https://openaccess.thecvf.com/content_cvpr_2018/html/Kostrikov_Surface_Networks_CVPR_2018_paper.html (visited on 01/16/2024).

[32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." In: *Advances in Neural Information Processing Systems*. Vol. 60. 6. New York, NY, USA: Association for Computing Machinery, May 2017, pp. 84–90. DOI: 10.1145/3065386.

[33] H. W. Kuhn. "The Hungarian Method for the Assignment Problem." In: *Naval Research Logistics Quarterly* 2.1-2 (1955), pp. 83–97. ISSN: 1931-9193. DOI: 10.1002/nav.3800020109.

[34] Y. LeCun et al. "Backpropagation Applied to Handwritten Zip Code Recognition." In: *Neural Computation* 1.4 (Dec. 1989), pp. 541–551. ISSN: 0899-7667. DOI: 10.1162/neco.1989.1.4.541.

[35] Yuwei Li et al. "NIMBLE: A Non-Rigid Hand Model with Bones and Muscles." In: *ACM Transactions on Graphics* 41.4 (July 2022), pp. 1–16. ISSN: 0730-0301, 1557-7368. DOI: 10.1145/3528223.3530079.

[36] Yuwei Li et al. *PIANO: A Parametric Hand Bone Model from Magnetic Resonance Imaging*. Aug. 2021. DOI: 10.24963/ijcai.2021/113.

[37] Isaak Lim et al. *A Simple Approach to Intrinsic Correspondence Learning on Unstructured 3D Meshes*. 2019. DOI: 10.1007/978-3-030-11015-4_26.

[38] UltraLeap Limited. *Leap Concepts - Ultraleap Documentation*. URL: https://docs.ultraleap.com/api-reference/tracking-api/leapc-guide/leap-concepts.html (visited on 02/12/2024).

[39] C. Karen Liu. "Dextrous Manipulation from a Grasping Pose." In: *ACM Transactions on Graphics* 28.3 (July 27, 2009), 59:1–59:6. ISSN: 0730-0301. DOI: 10.1145/1531326.1531365.

[40] Nadia Magnenat-Thalmann and Daniel Thalmann. "Virtual Humans: Thirty Years of Research, What Next?" In: *The Visual Computer* 21.12 (Dec. 1, 2005), pp. 997–1015. ISSN: 1432-2315. DOI: 10.1007/s00371-005-0363-6.

[41] Kien Mai Ngoc and Myunggwon Hwang. "Finding the Best k for the Dimension of the Latent Space in Autoencoders." In: *Computational Collective Intelligence*. Ed. by Ngoc Thanh Nguyen et al. Cham: Springer International Publishing, 2020, pp. 453–464. ISBN: 978-3-030-63007-2. DOI: 10.1007/978-3-030-63007-2_35.

[42] Jonathan Masci et al. *Geodesic Convolutional Neural Networks on Riemannian Manifolds*. 2015. DOI: 10.1109/ICCVW.2015.112.

[43] *Mickey Mouse*. In: *Wikipedia*. Feb. 20, 2024. URL: https://en.wikipedia.org/w/index.php?title=Mickey_Mouse&oldid=1209130613#cite_ref-54 (visited on 02/20/2024).

[44] Federico Monti et al. *Geometric Deep Learning on Graphs and Manifolds Using Mixture Model CNNs*. July 2017. DOI: 10.1109/CVPR.2017.576. URL: https://doi.ieeecomputersociety.org/10.1109/CVPR.2017.576.

[45] Gyeongsik Moon, Takaaki Shiratori, and Kyoung Mu Lee. *DeepHandMesh: A Weakly-supervised Deep Encoder-Decoder Framework for High-fidelity Hand Mesh Modeling*. Aug. 18, 2020. DOI: 10.1007/978-3-030-58536-5_26.

[46] James Munkres. "Algorithms for the Assignment and Transportation Problems." In: *Journal of the Society for Industrial and Applied Mathematics* 5.1 (1957), pp. 32–38. ISSN: 0368-4245. URL: https://www.jstor.org/stable/2098689.

[47] Brady Neal et al. *A Modern Take on the Bias-Variance Tradeoff in Neural Networks*. Dec. 18, 2019. DOI: 10.48550/arXiv.1810.08591.

[48] Keiron O'Shea and Ryan Nash. *An Introduction to Convolutional Neural Networks*. Dec. 2, 2015. DOI: 10.48550/arXiv.1511.08458. preprint.

[49] *ONNX*. URL: https://onnx.ai/ (visited on 02/27/2024).

[50] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[51] Akila Pemasiri et al. "Accurate 3D Hand Mesh Recovery from a Single RGB Image." In: *Scientific Reports* 12.1 (June 30, 2022), p. 11043. ISSN: 2045-2322. DOI: 10.1038/s41598-022-14380-x.

[52] Anurag Ranjan et al. *Generating 3D Faces Using Convolutional Mesh Autoencoders*. July 31, 2018. DOI: 10.1007/978-3-030-01219-9_43.

[53] Anke Verena Reinschluessel et al. "Virtual Reality for User-Centered Design and Evaluation of Touch-free Interaction Techniques for Navigating Medical Images in the Operating Room." In: *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. CHI EA '17. New York, NY, USA: Association for Computing Machinery, May 6, 2017, pp. 2001–2009. ISBN: 978-1-4503-4656-6. DOI: 10.1145/3027063.3053173.

[54] Javier Romero, Dimitrios Tzionas, and Michael J. Black. "Embodied Hands: Modeling and Capturing Hands and Bodies Together." In: *ACM Transactions on Graphics, (Proc. SIGGRAPH Asia)*. 245:1–245:17 36.6 (Nov. 2017). URL: https://mano.is.tue.mpg.de/.

[55] Guillermo M. Rosa and María L. Elizondo. "Use of a Gesture User Interface as a Touchless Image Navigation System in Dental Surgery: Case Series Report." In: *Imaging Science in Dentistry* 44.2 (June 2014), pp. 155–160. ISSN: 2233-7822. DOI: 10.5624/isd.2014.44.2.155.

[56] Janis Roßkamp et al. "UnrealHaptics: Plugins for Advanced VR Interactions in Modern Game Engines." In: *Frontiers in Virtual Reality* 2 (Apr. 1, 2021), p. 640470. DOI: 10.3389/frvir.2021.640470.

[57] Murat Sever and Sevda Öğüt. "A Performance Study Depending on Execution Times of Various Frameworks in Machine Learning Inference." In: *2021 15th Turkish National Software Engineering Symposium (UYMS)*. 2021 15th Turkish National Software Engineering Symposium (UYMS). Nov. 2021, pp. 1–5. DOI: 10.1109/UYMS54260.2021.9659677.

[58] Dalwinder Singh and Birmohan Singh. "Investigating the Impact of Data Normalization on Classification Performance." In: *Applied Soft Computing* 97 (Dec. 1, 2020), p. 105524. ISSN: 1568-4946. DOI: 10.1016/j.asoc.2019.105524.

[59] Martin Skrodzki. "The K-d Tree Data Structure and a Proof for Neighborhood Computation in Expected Logarithmic Time." In: (Mar. 12, 2019). DOI: 10.48550/arXiv.1903.04936.

[60] Qingyang Tan et al. "Realtime Simulation of Thin-Shell Deformable Materials Using CNN-Based Mesh Embedding." In: *IEEE Robotics and Automation Letters* 5.2 (Apr. 2020), pp. 2325–2332. ISSN: 2377-3766. DOI: 10.1109/LRA.2020.2970624.

[61] N. Tomizawa. "On Some Techniques Useful for Solution of Transportation Network Problems." In: *Networks* 1.2 (1971), pp. 173–194. ISSN: 1097-0037. DOI: 10.1002/net.3230010206.

[62] Raoul Tubiana, Jean-Michel Thomine, and Evelyn Mackin. *Examination of the Hand and Wrist*. 2nd ed, repr. London: Informa Healthcare, 2007. 397 pp. ISBN: 978-1-85317-544-2.

[63] Ultraleap. *Leap Motion Controller*. URL: https://www.ultraleap.com/leap-motion-controller-overview/ (visited on 03/07/2024).

[64] Neural VFX and charlievfx. *GitHub - Basic-Unreal-Libtorch-Plugin: A "Hello World" for Running LibTorch inside Unreal Engine*. URL: https://github.com/NeuralVFX/basic-unreal-libtorch-plugin/tree/master (visited on 03/05/2024).

[65] Nkenge Wheatland et al. "State of the Art in Hand and Finger Modeling and Animation." In: *Computer Graphics Forum* 34.2 (2015), pp. 735–760. ISSN: 1467-8659. DOI: 10.1111/cgf.12595.

[66] Michael Wilkinson, Sean Brantley, and Jing Feng. "A Mini Review of Presence and Immersion in Virtual Reality." In: *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 65.1 (Sept. 1, 2021), pp. 1099–1103. ISSN: 1071-1813. DOI: 10.1177/1071181321651148.

[67] Ian H. Witten et al. "Chapter 2 - Input: Concepts, Instances, Attributes." In: *Data Mining (Fourth Edition)*. Ed. by Ian H. Witten et al. Fourth Edition. Morgan Kaufmann, 2017, pp. 56–57. ISBN: 978-0-12-804291-5. DOI: 10.1016/B978-0-12-804291-5.00002-7.

A Appendix

A.1 Content of memory stick

- a copy of this thesis
- the Unreal Engine project with the installed plugins
- the project of the wrapper plugin
- the NIMBLE and LEAP dataset used in this tesis
- the Python project used to create the neural network
- traces of the two pose encoders, with a latent space size of 64 and 128

A.2 Results of the parameter study

Table 2: Results of the parameter study for autoencoders

ID	bs	lr	lss	sql	out channels	pool steps	Parameters	Mean	Variance	Median	Avg. E. t.	Avg. Inf. t.
											seconds	ms
3	32	0.0001	128	13	[128, 128, 256, 512]	[1, 1, 1, 1]	62,604,003	50.797	166.187	23.024	20.67	-
4	32	0.0001	128	13	[128, 128, 256, 256, 512]	[1, 1, 1, 1, 1]	38,912,739	59.191	181.243	27.070	18.81	-
6	32	0.0001	128	13	[128, 128, 256, 256, 512]	[1, 1, 2, 2, 1]	18,254,051	72.116	203.364	33.676	15.43	-
5	32	0.0001	128	13	[128, 128, 256, 512]	[2, 2, 2, 2]	12,733,667	81.310	220.752	38.165	10.25	-
36	32	0.0003	64	9	[64, 128, 128, 256]	[1, 1, 1, 1]	15,308,255	93.065	230.170	45.230	7.61	0.99
46	32	0.0003	64	13	[64, 128, 128, 256]	[1, 1, 1, 1]	16,030,687	95.843	229.900	46.707	10.06	1.12
56	32	0.0003	64	15	[64, 128, 128, 256]	[1, 1, 1, 1]	16,391,903	100.813	220.755	49.786	11.27	1.23
39	32	0.0003	64	9	[128, 128, 256, 256, 512]	[1, 1, 1, 1, 1]	21,398,751	102.719	224.834	49.871	13.76	1.67
49	32	0.0003	64	13	[128, 128, 256, 256, 512]	[1, 1, 1, 1, 1]	24,416,479	107.162	216.395	54.372	18.65	2.32
58	32	0.0003	64	15	[64, 128, 128, 256, 256]	[1, 1, 1, 1, 1]	11,984,863	107.653	226.311	54.146	12.65	1.69
38	32	0.0003	64	9	[64, 128, 128, 256, 256]	[1, 1, 1, 1, 1]	10,114,783	107.772	237.942	52.802	8.41	1.12
48	32	0.0003	64	13	[64, 128, 128, 256, 256]	[1, 1, 1, 1, 1]	11,361,503	108.583	227.546	53.344	11.16	1.46
55	32	0.0003	64	15	[64, 64, 64, 128]	[1, 1, 1, 1]	7,589,343	110.705	255.488	54.278	6.46	0.99
45	32	0.0003	64	13	[64, 64, 64, 128]	[1, 1, 1, 1]	7,490,271	114.580	262.984	55.137	5.86	1.01
35	32	0.0003	64	9	[64, 64, 64, 128]	[1, 1, 1, 1]	7,292,127	115.411	262.548	57.126	4.59	0.99
59	32	0.0003	64	15	[128, 128, 256, 256, 512]	[1, 1, 1, 1, 1]	25,925,343	121.976	225.902	63.004	21.04	2.62
47	32	0.0003	64	13	[64, 64, 64, 128, 128]	[1, 1, 1, 1, 1]	4,729,695	128.511	281.905	63.886	6.26	1.00
6	32	0.001	64	9	[64, 128, 128, 256]	[1, 1, 1, 1]	15,308,255	129.450	231.666	67.759	7.59	0.98
57	32	0.0003	64	15	[64, 64, 64, 128, 128]	[1, 1, 1, 1, 1]	4,894,303	132.642	283.162	65.027	6.89	0.99
37	32	0.0003	64	9	[64, 64, 64, 128, 128]	[1, 1, 1, 1, 1]	4,400,479	137.222	282.735	67.168	4.89	1.00
5	32	0.001	64	9	[64, 64, 64, 128]	[1, 1, 1, 1]	7,292,127	138.574	254.748	69.100	4.59	1.00
44	32	0.0003	32	13	[128, 128, 256, 256, 512]	[1, 1, 1, 1, 1]	17,172,767	240.621	397.300	116.850	18.64	2.28
41	32	0.0003	32	13	[64, 128, 128, 256]	[1, 1, 1, 1]	9,245,727	241.927	424.817	116.598	9.99	1.21
34	32	0.0003	32	9	[128, 128, 256, 256, 512]	[1, 1, 1, 1, 1]	14,155,039	242.291	415.091	116.571	13.72	1.65
51	32	0.0003	32	15	[64, 128, 128, 256]	[1, 1, 1, 1]	9,606,943	242.957	420.311	116.838	11.26	1.22
31	32	0.0003	32	9	[64, 128, 128, 256]	[1, 1, 1, 1]	8,523,295	246.781	425.797	119.108	7.57	0.98

54	32	0.0003	32	15	[128, 128, 256, 256, 512]	[1, 1, 1, 1, 1]	18,681,631	253.364	403.631	127.543	21.05	2.63
33	32	0.0003	32	9	[64, 128, 128, 256, 256]	[1, 1, 1, 1, 1]	6,491,935	259.574	437.403	126.480	8.38	1.21
0	32	0.001	32	9	[64, 64, 64, 128]	[1, 1, 1, 1]	3,898,655	268.463	435.127	138.217	4.61	0.98
1	32	0.001	32	9	[64, 128, 128, 256]	[1, 1, 1, 1]	8,523,295	271.027	421.862	138.689	7.59	0.99
50	32	0.0003	32	15	[64, 64, 64, 128]	[1, 1, 1, 1]	4,195,871	271.314	453.167	137.468	6.45	0.98
10	32	0.001	32	13	[64, 64, 64, 128]	[1, 1, 1, 1]	4,096,799	274.119	428.246	142.599	5.85	0.98
40	32	0.0003	32	13	[64, 64, 64, 128]	[1, 1, 1, 1]	4,096,799	274.495	458.300	136.426	5.92	0.99
28	32	0.001	64	15	[64, 128, 128, 256, 256]	[1, 1, 1, 1, 1]	11,984,863	276.583	351.819	166.423	12.43	1.52
52	32	0.0003	32	15	[64, 64, 64, 128, 128]	[1, 1, 1, 1, 1]	3,081,887	284.830	467.663	144.516	6.90	1.00
30	32	0.0003	32	9	[64, 64, 64, 128]	[1, 1, 1, 1]	3,898,655	289.906	478.328	144.345	4.56	0.99
42	32	0.0003	32	13	[64, 64, 64, 128, 128]	[1, 1, 1, 1, 1]	2,917,279	294.459	482.644	147.839	6.32	0.98
32	32	0.0003	32	9	[64, 64, 64, 128, 128]	[1, 1, 1, 1, 1]	2,588,063	298.132	475.656	153.160	4.88	0.99
29	32	0.001	64	15	[128, 128, 256, 256, 512]	[1, 1, 1, 1, 1]	25,925,343	305.558	379.528	181.998	20.58	2.61
4	32	0.001	32	9	[128, 128, 256, 256, 512]	[1, 1, 1, 1, 1]	14,155,039	343.652	474.020	192.429	13.80	1.63
14	32	0.001	32	13	[128, 128, 256, 256, 512]	[1, 1, 1, 1, 1]	17,172,767	387.939	503.723	225.004	18.36	2.34
2	32	0.001	32	9	[64, 64, 64, 128, 128]	[1, 1, 1, 1, 1]	2,588,063	394.080	504.432	234.320	4.88	0.98
24	32	0.001	32	15	[128, 128, 256, 256, 512]	[1, 1, 1, 1, 1]	18,681,631	1328.710	1489.482	758.377	20.57	2.60
19	32	0.001	64	13	[128, 128, 256, 256, 512]	[1, 1, 1, 1, 1]	24,416,479	10517.672	12989.236	4686.535	18.04	2.40
11	32	0.001	32	13	[64, 128, 128, 256]	[1, 1, 1, 1]	9,245,727	10528.438	12886.705	4813.941	9.88	1.21
25	32	0.001	64	15	[64, 64, 64, 128]	[1, 1, 1, 1]	7,589,343	10536.466	12863.673	4815.466	6.41	0.98
21	32	0.001	32	15	[64, 128, 128, 256]	[1, 1, 1, 1]	9,606,943	10543.411	12852.186	4806.941	11.05	1.22
9	32	0.001	64	9	[128, 128, 256, 256, 512]	[1, 1, 1, 1, 1]	21,398,751	10548.855	12840.901	4814.374	13.26	1.76
8	32	0.001	64	9	[64, 128, 128, 256, 256]	[1, 1, 1, 1, 1]	10,114,783	10561.388	12923.341	4802.612	8.45	-
16	32	0.001	64	13	[64, 128, 128, 256]	[1, 1, 1, 1]	16,030,687	10570.127	12973.792	4710.657	9.77	1.13
3	32	0.001	32	9	[64, 128, 128, 256, 256]	[1, 1, 1, 1, 1]	6,491,935	10578.381	12827.492	4803.844	8.33	1.20
18	32	0.001	64	13	[64, 128, 128, 256, 256]	[1, 1, 1, 1, 1]	11,361,503	10579.051	12867.471	4775.597	10.90	1.53
23	32	0.001	32	15	[64, 128, 128, 256, 256]	[1, 1, 1, 1, 1]	8,362,015	10583.633	12823.625	4851.202	12.23	1.69
7	32	0.001	64	9	[64, 64, 64, 128, 128]	[1, 1, 1, 1, 1]	4,400,479	10596.085	12816.618	4825.218	4.89	0.99
12	32	0.001	32	13	[64, 64, 64, 128, 128]	[1, 1, 1, 1, 1]	2,917,279	10602.024	12842.198	4800.272	6.33	-
43	32	0.0003	32	13	[64, 128, 128, 256, 256]	[1, 1, 1, 1, 1]	7,738,655	10636.270	12908.559	4779.136	11.17	-
53	32	0.0003	32	15	[64, 128, 128, 256, 256]	[1, 1, 1, 1, 1]	8,362,015	10643.762	12988.984	4748.883	12.59	-
27	32	0.001	64	15	[64, 64, 64, 128, 128]	[1, 1, 1, 1, 1]	4,894,303	10668.433	12988.893	4985.308	6.87	0.99
20	32	0.001	32	15	[64, 64, 64, 128]	[1, 1, 1, 1]	4,195,871	10675.166	12871.173	4826.085	6.51	0.99

15	32	0.001	64	13	[64, 64, 64, 128]	[1, 1, 1, 1]	7,490,271	10683.676	12815.383	4880.034	5.85	0.99
22	32	0.001	32	15	[64, 64, 64, 128, 128]	[1, 1, 1, 1, 1]	3,081,887	10690.227	12948.309	4997.000	6.93	-
13	32	0.001	32	13	[64, 128, 128, 256, 256]	[1, 1, 1, 1, 1]	7,738,655	10693.675	12886.409	4828.702	11.07	-
26	32	0.001	64	15	[64, 128, 128, 256]	[1, 1, 1, 1]	16,391,903	10869.618	13081.866	5112.475	11.17	-
17	32	0.001	64	13	[64, 64, 64, 128, 128]	[1, 1, 1, 1, 1]	4,729,695	11074.187	14002.287	4821.220	6.29	-

Table 3: Results of the parameter study for pose encoders

ID	bs	lr	ls	sql	out channels	pool steps	Parameters	Mean	Variance	Median	Avg. Epoch t
0	32	0.0001	32	13	[128, 128, 256, 256, 512]	[1, 1, 1, 1, 1, 1]	10,348,451	1635.564	1454.345	1254.538	11.56
1	32	0.0001	64	13	[64, 128, 128, 256]	[1, 1, 1, 1, 1]	8,501,019	1653.109	1591.774	1227.110	6.81
2	32	0.0001	64	13	[62, 124, 248, 248, 248]	[1, 1, 2, 4, 1, 1]	3,219,551	1481.690	1221.817	1172.137	8.44
3	32	0.0001	128	13	[128, 128, 256, 512]	[1, 1, 1, 1, 1]	33,123,043	1547.427	1331.005	1200.830	12.63
4	32	0.0001	128	13	[128, 128, 256, 256, 512]	[1, 1, 1, 1, 1, 1]	21,228,003	1505.942	1278.192	1179.067	11.58
5	32	0.0001	128	13	[128, 128, 256, 512]	[2, 2, 2, 2, 1]	8,090,851	1490.508	1279.376	1165.228	7.25
6	32	0.0001	128	13	[128, 128, 256, 256, 512]	[1, 1, 2, 2, 1, 1]	10,858,467	1417.518	1145.121	1138.088	9.91
0	32	0.001	32	13	[128, 128, 256, 256, 512]	[1, 1, 1, 1, 1, 1]	10,348,451	1205.871	829.131	1023.735	54.93
1	32	0.001	64	13	[64, 128, 128, 256]	[1, 1, 1, 1, 1]	8,501,019	1180.967	816.965	1003.288	30.71
2	32	0.001	64	13	[62, 124, 248, 248, 248]	[1, 1, 2, 4, 1, 1]	3,219,551	5181.328	7067.889	2435.251	39.63
3	32	0.001	128	13	[128, 128, 256, 512]	[1, 1, 1, 1, 1]	33,123,043	1169.857	795.825	997.317	61.44
4	32	0.001	128	13	[128, 128, 256, 256, 512]	[1, 1, 1, 1, 1, 1]	21,228,003	1166.495	786.613	995.614	54.87
5	32	0.001	128	13	[128, 128, 256, 512]	[2, 2, 2, 2, 1]	8,090,851	2257.518	3080.999	1341.007	33.51
6	32	0.001	128	13	[128, 128, 256, 256, 512]	[1, 1, 2, 2, 1, 1]	10,858,467	3526.446	6535.099	1435.018	46.95

Universität Bremen

Offizielle Erklärungen von

Nachname: Höring Vorname: Thomas

Matrikelnr.: 4381005

A) Eigenständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.
Alle Teile meiner Arbeit, die wortwörtlich oder dem Sinn nach anderen Werken entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht. Gleiches gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet.
Die Arbeit wurde in gleicher oder ähnlicher Form noch nicht als Prüfungsleistung eingereicht.
Die elektronische Fassung der Arbeit stimmt mit der gedruckten Version überein.
Mir ist bewusst, dass wahrheitswidrige Angaben als Täuschung behandelt werden.

B) Erklärung zur Veröffentlichung von Bachelor- oder Masterarbeiten

Die Abschlussarbeit wird zwei Jahre nach Studienabschluss dem Archiv der Universität Bremen zur dauerhaften Archivierung angeboten. Archiviert werden:
1) Masterarbeiten mit lokalem oder regionalem Bezug sowie pro Studienfach und Studienjahr 10 % aller Abschlussarbeiten
2) Bachelorarbeiten des jeweils ersten und letzten Bachelorabschlusses pro Studienfach u. Jahr.

☑ Ich bin damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

☑ Ich bin damit einverstanden, dass meine Abschlussarbeit nach 30 Jahren (gem. §7 Abs. 2 BremArchivG) im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

☐ Ich bin nicht damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

Mit meiner Unterschrift versichere ich, dass ich die oben stehenden Erklärungen gelesen und verstanden habe. Mit meiner Unterschrift bestätige ich die Richtigkeit der oben gemachten Angaben.

20.03.2024, Bremen
Datum, Ort

Thomas Höring
Unterschrift