FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Edge flow correction of 3D models using neural networks
Student:	Bc. Anh Viet Tran
Supervisor:	Ing. Tomáš Nováček
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2024/2025

Instructions

Often, a 3D artist wants to lower the number of polygons in the 3D model. There are analytical algorithms which can effectively do the job, but these algorithms do not take into consideration the edge flow of the mesh, which is critical for animators. The goal of the thesis is to propose a solution to correct these errors, which is a topic previously not researched by the scientific community.

The goals of the thesis:
1) Analyze the current state of edge flow correction, focus on the available tools and the workflow.
2) Create a dataset to be used for edge flow corrections.
3) Propose a deep learning architecture for edge flow corrections.
4) Train the proposed architecture on the created dataset and evaluate it in terms of precision and accuracy.

Proposed literature:
Ju, Tao & Zhou, Qian-Yi & Hu, Shi-Min. (2007). Editing the Topology of 3D Models by Sketching. ACM Trans. Graph.. 26. 42. 10.1145/1276377.1276430.

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 6 June 2023 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Edge flow correction of 3D models using neural networks

Bc. Viet Anh Tran

Department of Applied Mathematics
Supervisor: Ing. Tomáš Nováček

May 8, 2024

Acknowledgements

First and foremost I'd like to thank my mom for her endless support in my life journey. I'd also like to thank my other parents, Marcela and Ivan, for guiding me towards academics in very early age and helping my mom in the toughest times. Finally I'd like to thank my supervisor Ing. Tomáš Nováček for his support and guidance on and off my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 8, 2024 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Tran, Viet Anh. *Edge flow correction of 3D models using neural networks.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Abstrakt

Edge flow je základní koncept v 3D modelovaní. Hraje významnou roli v přirozené 3D deformaci, převážně využívané v animacích. Ne vždy má 3D mesh optimální edge flow a existuje mnoho nástrojů k jeho zlepšení, které jsou všechny založeny na analytickém přístupu. V této práci zkoumáme možnosti stochastického přístupu k tomuto optimalizačnímu problému tím, že jsme problém přeformulovali na rekonstruční úlohu. Poukázali jsme na limitace Chamfer vzdálenosti, které způsobuje nepřesnosti v mnohých publikovaných prací v oblasti rekonstrukce tvaru. Navrhli jsme predikovat tvary na základě odchylek bodů s čímž jsme i diskutovali různé způsoby jak tyto odchylky předpočítat pro trénovací potřeby, kde jsme se ve výsledku obrátili ke stochastickému způsobu výpočtu. Naimplementovali jsme kompletní řešení pro rekonstrukci tvaru, ale neuspěli v trénování samotného modelu. Přes změny v architektuře, vektoru příznaků nebo aplikování trénovací technik a optimalizování parametrů, výsledek trénování byl stále stejně neúspěšný. Došli jsme k závěru, že náš model nebyl dostatečně silný na to se naučit klíčové příznaky tvaru.

Klíčová slova optimalizace meshe, edge flow, topologie, grafové neuronové sítě

Abstract

Edge flow is a fundamental concept in 3D modeling, playing a pivotal role in natural 3D model deformations mostly used in animation. A 3D mesh does not always possess optimized edge flow. Thus, many tools exist to optimize it, among which all are analytically based solutions. This thesis explores a stochastic approach to the optimization problem while attempting to reframe the problem into a reconstruction task. We point out a common issue in regards to Chamfer distance. The limitation of Chamfer distance hinders the quality of shape reconstruction in most of the published researches. We proposed predicting offsets of the mesh points while exploring how to compute the ground truth for these offsets. We discuss several solutions and their limitations while resorting to stochastic computational solutions. We fully implemented the prediction/reconstruction pipeline but failed to train the model to predict point offsets. Despite many attempts to fix the training by changing the architecture, applying deep learning techniques, tuning the optimizer's hyperparameters, or extending the feature vector, the training result remained defective. We concluded that the model was not powerful enough to learn key shape features.

Keywords mesh optimization, edge flow, topology, graph neural network

Contents

List of Figures

Introduction

Computer graphics and 3D modeling is a fundamental field in our constantly increasing digital world. Its application in engineering, science, art, product visualization, and most notably, entertainment has pushed this field with a constant demand for better visual quality and performance optimization.

3D models create the very core of any 3D visualization, bringing objects to computer-generated 3D space. The 3D model defines the shape of an object and can be further improved to closely represent real-world objects by applying textures, lighting, and many more techniques. Models are typically created in 3D modeling software by an artist or by 3D scanning. 3D scanning can often bring too many details in terms of polygon numbers – a shape that could be perfectly recognizable by ten polygons would now be described by thousands. The side effect of things is big hardware requirements.

Lowering the number of polygons is a usual practice done either manually or one by many developed tools over the years. Some tools do not consider any edge flow, a critical feature particularly important for character modeling and animation. Where some require great deal of manual work to reach optimal edge flow quality. There has been a great advancement in this field and highly optimal analytical solution has been already developed, leading in quality and speed. Edge flow allows smooth deformation. In terms of animation, we can look at muscle movement – optimal edge avoids wrinkled defects on the character model, and the skin would stretch and contract in a way as we are used to seeing in the real world.

This thesis aims to explore the final step in polygon reduction workflow when using tools. We present a neural network taking a 3D model and outputting an improved model in terms of edge flow quality, approximating similar polygon quantity.

In Chapter Neural Networks, we explore basics of neural networks, introduce its terminology and several commonly used network architectures, while also introducing their more advance variantion.

In Chapter Related Work, we take a look at number of researches related to our topics. We also point out some of their limitations.

In Chapter Implementation, we describe key ideas behind our proposed solution and the reasoning behind them. We explore encountered challenges how we attempted to solve them.

In Chapter New State of the Art, we will take a closer look at the research that was published during the writing of this thesis. The findings can be viewed as the new state of the art in shape reconstruction, and we discuss whether the new findings are applicable to the topic we are trying to solve.

In Conclusion, we evaluate the results of the work and suggest possible improvements for future works.

Chapter 1

Neural Networks

An artificial neural network (ANN), first introduced by Warren McCulloch and Walter Pitts in „A logical calculus of the ideas immanent in nervous activity“ published in 1943 [19], is a mathematical model based on biological neural networks. It carries the ability to learn and correct errors from previous experience [20, 21].

The ANN has gained popularity in recent years with still increasing advancements in technology and availability of training data. ANN now becomes a default solutions for complex tasks previously thought to be unsolvable by computers [22].

This chapter will briefly introduce different types of neural units and their activation functions, along with some commonly used network architectures.

1.1 Artificial Neuron

Artificial neurons are units of ANN, mimicking behavior of biological neurons. Just as biological neurons, it can receive as well as pass information to other neurons.

1.1.1 Perceptron

Perceptron, developed by Frank Rosenblatt in 1958 [23], is the simplest class of artificial neurons.

Perceptron takes several binary inputs in form of a vector $\vec{x} = (x_1, x_2, ..., x_n)$, and outputs a single binary number. Perceptron uses real numbers called *weights*, assigned to each edge, vector $\vec{w} = (w_1, w_2, ..., w_n)$, to express the importance of respected input edges.

A *step function* calculates the perceptron's output. The function output is either 0 or 1 determined by whether its weighted sum $\alpha = \sum_i x_i w_i$ is less or greater than its *threshold* value, a real number, usually represented as an incoming edge with a negative weight -1 [1].

$$output = \begin{cases} 1, & \text{if } \alpha \geq threshold \\ 0, & \text{if } \alpha < threshold \end{cases} \tag{1.1}$$

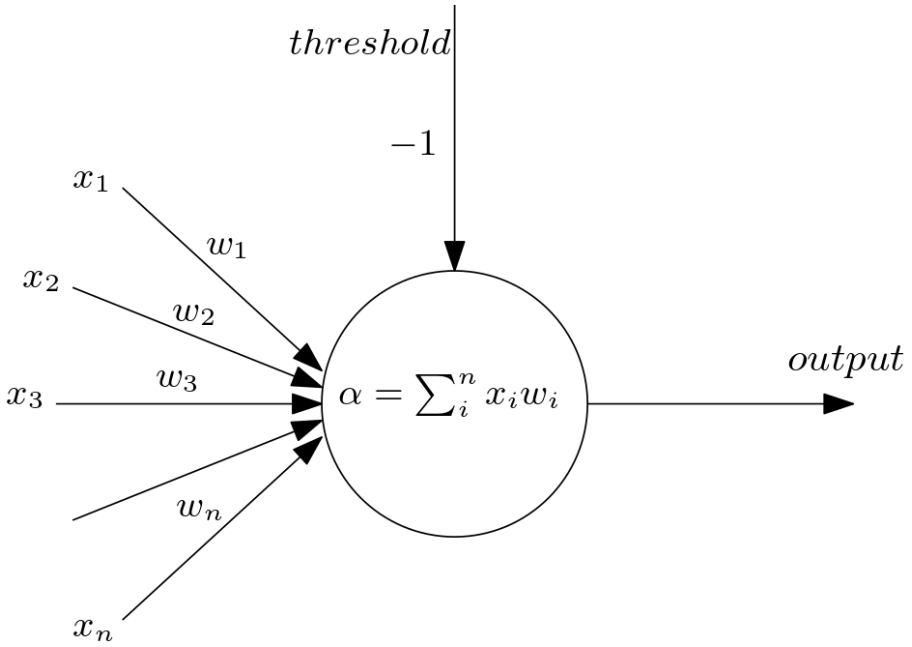


Figure 1.1: Perceptron [1]

1.1.2 Sigmoid Neuron

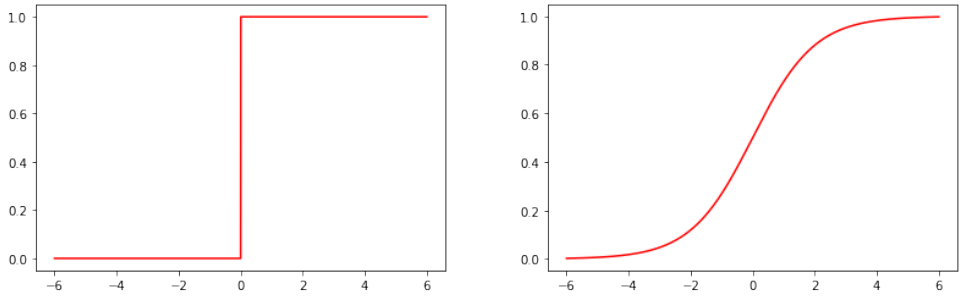
Sigmoid neuron, similarly to perceptron, has vector inputs $\vec{x}$ and weights. The difference is in the output value and its calculation. Instead of perceptron's binary output 0 or 1, a sigmoid neuron outputs a real number between 0 and 1 using a *sigmoid function* [24, 25, 1].

$$\sigma(\alpha) = \frac{1}{1 + e^{-\alpha}} \tag{1.2}$$

As shown in Figure 1.2, the sigmoid function (1.2b) is a smoothed-out step function (1.2a).

1.2 Activation Function

An artificial neuron's activation function defines neuron's output value for given inputs, commonly being $f : \mathbb{R} \to \mathbb{R}$ [26]. An important trait of many activation functions is their differentiability, allowing them to be used for *Backpropagation*, ANN training algorithm which we will explore later in the



(a) Step function

(b) Sigmoid function

Figure 1.2: Comparison between step function and sigmoid function

chapter. The activation function needs to have a derivative that does not saturate when headed towards 0 or explode when headed towards inf [1].

For such reasons, step function or any linear function are unsuitable for ANN.

1.2.1 Sigmoid Function

The sigmoid function is often used in ANN as an alternative to the step function. A popular choice of the sigmoid function is a *logistic sigmoid*, output value ranging between 0 and 1.

$$\sigma(\alpha) = \frac{1}{1 + e^{-\alpha}} = \frac{e^x}{1 + e^x} \tag{1.3}$$

One of the reasons for its popularity is the simplicity of its derivative calculation:

$$\frac{d}{dx}\sigma(\alpha) = \frac{e^x}{(1 + e^x)^2} = \sigma(x)(1 - \sigma(x)) \tag{1.4}$$

Its disadvantages is the *vanishing gradient*. A problem where if given a very high or very low input values, the prediction would stay almost the same. Possibly resulting in training complications or performance issues [27, 1].

1.2.2 Hyperbolic Tangent

Hyperbolic tangent is similar to logistic sigmoid function with a key difference in its output, ranging between -1 and 1.

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{1.5}$$

It shares the sigmoid's simple calculation of its derivative.

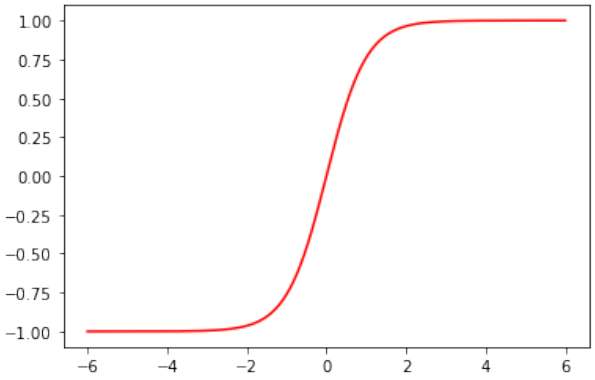


Figure 1.3: Hyperbolic tangent [1]

$$\frac{d}{dx}\tanh(x) = 1 - \frac{(e^x - e^{-x})^2}{(e^x + e^{-x})^2} = 1 - \tanh^2(x) \tag{1.6}$$

By being only moved and scaled version of the sigmoid function, hyperbolic tangent shares not only sigmoid's advantages but also its disadvantages [26, 1].

1.2.3 Rectified Linear Unit

The output of the Rectified Linear Unit (ReLU) is defined as:

$$f(x) = max(0, x) \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases} \tag{1.7}$$

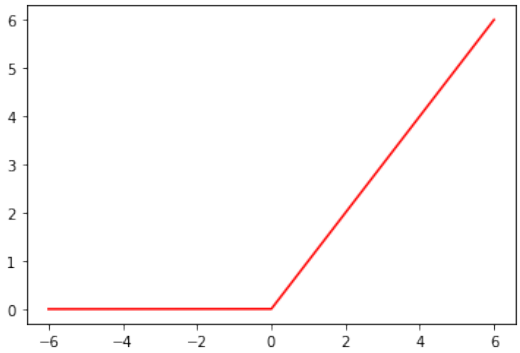


Figure 1.4: Rectified Linear Unit [1]

ReLU's popularity is mainly due to its computational efficiency [27]. Its disadvantages begin to show themselves once inputs approach zero or to a negative number. Causing the so-called dying ReLu issue, where the network is unable to learn anymore. There are many variations of ReLu to this date, e.g., Leaky ReLU, Parametric ReLU, ELU, ...

1.2.4 Softmax

Softmax separates itself from all the previously mentioned functions by its ability to handle multiple input values in the form of a vector $\vec{x} = (x_1, x_2, ..., x_n)$ and output for each x_i defined as:

$$\sigma(x_i) = \frac{e_i^x}{\sum_{j=1}^{n} e_j^x} \tag{1.8}$$

Output is normalized probability distribution, ensuring $\sum_i \sigma(x_i) = 1$ [28]. It is being used as the last activation function of ANN, normalizing the network's output into n probability groups.

1.3 Neural Networks Learning

We will now discuss how the neural networks learn and propagate information on very simple example of feed-forward network (FFN), the first invented ANN and the simplest variation of an ANN. Its name comes from the way how information flows through the network. The data flows in one direction, oriented from the *input layer* to the *output layer*, without cycles. The input layer takes input data, vector $\vec{x}$, producing $\hat{y}$ at the output layer [29].

FFN contains several hidden layers of various widths but it can also have no hidden layers at all. By having no back-loops, FFN generally minimizes error, computed by *cost function*, in its prediction by using the *backpropagation* algorithm to update its weight values [30, 28].

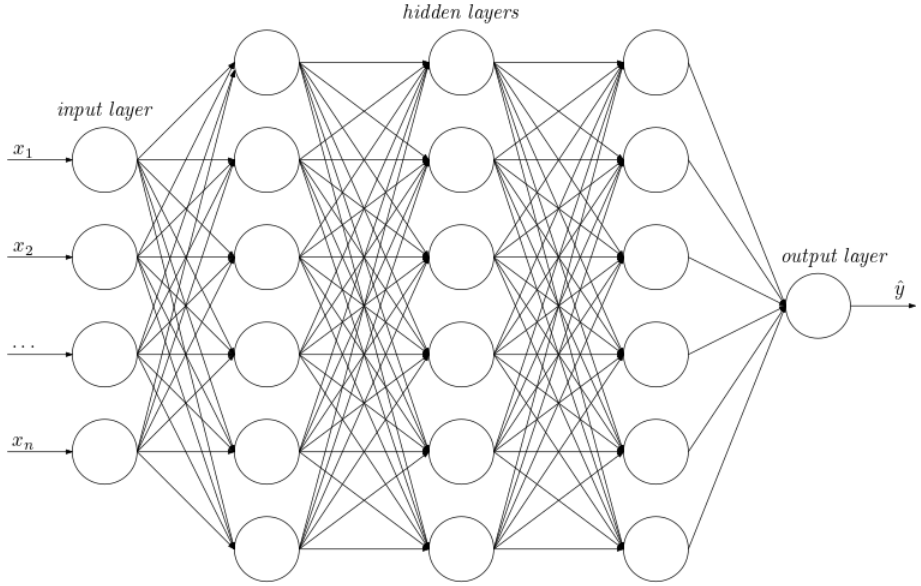


Figure 1.5: Fully connected Feed-forward Neural Network [1]

1.3.1 Cost Function

Cost function $C(\vec{w})$ is used in ANN's training process. It takes all weights and biases of an ANN as its input, in the form of a vector $\vec{w}$ and calculates a single real number expressing ANN's error in prediction [31]. Higher number expresses poor prediction and as the number gets lower the ANN's output gets closer to the correct result. The main goal of training is to minimize the cost function.

1.3.2 Backpropagation

Backpropagation, short of backward propagation of errors, is a widely used algorithm in training FFN using *gradient descent* to find a local minimum of a cost function and update ANN's weights [32].

The gradient of a multi-variable function provides us with the direction of the gradient ascent, where we should step to rapidly increase the output and find the local maximum. Conversely, the negative of the gradient points towards the local minimum.

It is common practice to split training samples into small *batches* of size n. For each sample in the batch, we will calculate a gradient descent and use their average gradient descent to update the network's weights. This average gradient descent indicates the adjustments that need to be made to the weights so that the artificial neural network (ANN) moves closer to the correct results [32].

$$-\gamma \nabla C(\vec{w_i}) + \vec{w_i} \rightarrow \vec{w}_{i+1} \tag{1.9}$$

$\vec{w_i}$ is weights of the network at the current state (batch), $\vec{w_{i+1}}$ is updated weights, γ is the learning rate and $-\nabla C(\vec{w_i})$ is the gradient descent.

1.4 Types of Neural Networks

To this day, there are many types and variations of ANN, each with its structure and use cases. Here we will not discuss all of them, but we will briefly explore convolutional neural networks and its more advanced variation, graph neural network. We will also briefly introduce recurrent neural networks and LSTM cells.

1.4.1 Convolutional Neural Networks

The primary objective of a Convolutional Neural Network (CNN) is to enable a computer to identify images and objects, making it ideal for image classification and object recognition tasks.

CNNs are based on the biological processes in the human brain and its connectivity patterns resemble those of the human visual cortex. However,

images are perceived differently by the human brain and a computer, with the latter interpreting images as arrays of numbers. As a result, CNNs are designed to work with two-dimensional image arrays, although they can also work with one-dimensional or three-dimensional arrays [33].

CNN is a variation of FNN [31]. It typically consists of an input layer, followed by multiple hidden layers, including several *convolutional layers* and *pooling layers*, and concluding with an output layer.

1.4.1.1 Convolutional Layer

The convolutional layer's goal is to extract key features from the input image by passing a matrix known as a *kernel* over the input image abstracted into a matrix [34].

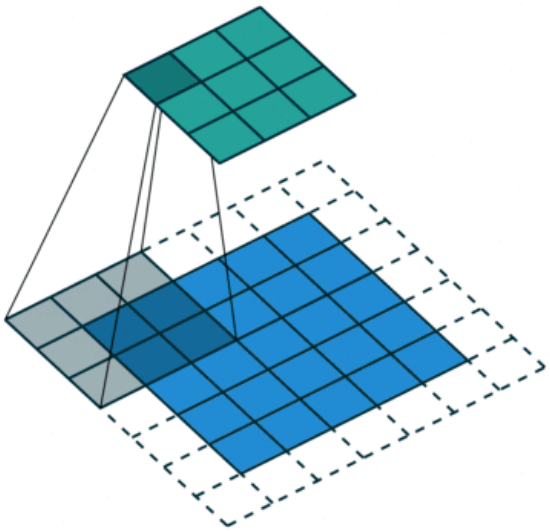


Figure 1.6: Convolution of an 5x5x1 image with 3x3x1 kernel [2]

The outcome of a convolution operation can be either reduced or increased in size. If the size is reduced, it is referred to as *valid padding*. For example, a convolution operation on an 8x8 input image would result in a 6x6 convoluted feature. On the other hand, if the size remains the same or is increased, it is referred to as *same padding* [2].

1.4.1.2 Pooling Layer

Like the convolutional layer, the pooling layer reduces the size of the convolved feature to reduce computational power needed for data processing. It also extracts dominant features that are invariant to rotation and position, making it beneficial for training the model effectively [2].

There are two types of pooling: max pooling and average pooling. Max pooling returns the maximum value from the portion of the image covered by the kernel, acting as a noise suppressant by removing noisy activations and performing de-noising and dimensionality reduction. Average pooling returns the average of all values in the same portion, reducing dimensions as a noise suppression mechanism. It is worth noting that max pooling performs better [2].

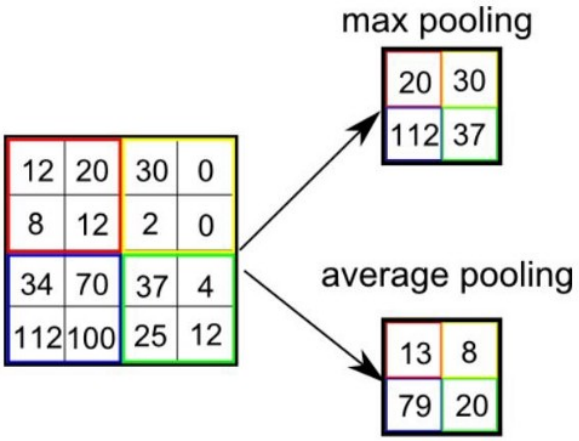


Figure 1.7: Types of pooling [2]

1.4.2 Graph Neural Networks

Graphs are a powerful data structure that describes and analyzes entities with relations or interactions. They consist of vertices (also called nodes) and edges connecting vertices, often representing some connection/relation between vertices. Edges can be either undirected or directed. Graphs hold many properties and can express many types of relations, thus often making them the best choice for problem representation. Still, today's deep learning modern toolbox is specialized for simple data types, e.g., grids for images, sequences for text or speech. These data structures have spatial locality. The grid size or sequence length is fixed and can be resized. We can also determine the starting position and ending position. Graph problems are, on the other hand, much more challenging to process as they have arbitrary size and complex topological structure (i.e., no spatial locality like grids). Graphs also have no fixed node ordering or reference point compared to grids or sequences. This is a direct result of graph isomorphism, which can be viewed as a computational challenge when it comes to identifying graph topologies and their similarity, one input can have many equivalent representations. For such, Franco Scarselli et al. introduced graph neural network (GNN) [3, 4].

GNN is designed similarly to CNN. As previously noted, CNN operates on images. Given an image, a rectangular grid, the convolutional layer takes a

subpart of the image, applies a function to it, and produces a new part, a new pixel. This is iterated for the whole image. What actually happened is the new pixel resulted in aggregated information from neighbors and itself. This cannot be easily applied to graphs as they have no spatial locality and no fixed node ordering. As implied, a GNN design stands on passing and aggregating information from neighbors [4].

1.4.2.1 Node embedding

Graphs require a concept called node embedding. The general idea is to map nodes to a lower dimensional embedding space, where similar nodes in the embedding space approximate similarity in the graph network. For example, we can map a 3D vector to a 2D vector. Node embedding is useful for learning node representations and similarities and can be trained on graphs with arbitrary structures [3, 4].

Nodes have embeddings at each layer. Taken node v, its layer-0 embedding would be its feature vector x_v. If we want layer-1 embedding, we will explore node v's neighbors. These neighbors are so-called one hop away from our original vector v. We take the feature vector of these nodes and aggregate the information into one single x_v feature vector. Layer-k would get information from nodes that are k hops away [3].

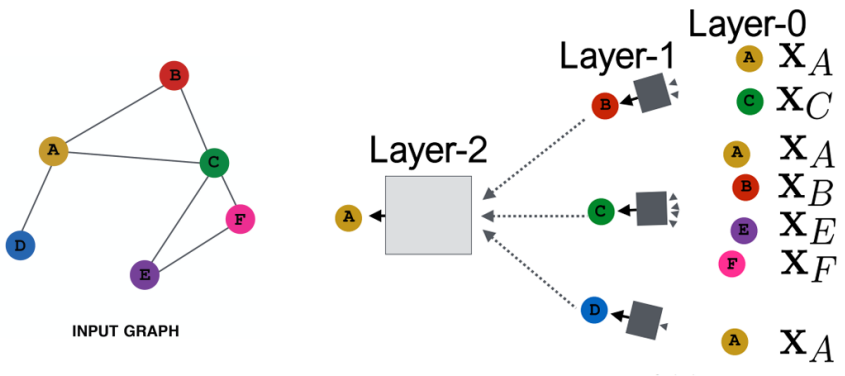


Figure 1.8: Layer-2 embedding applied on node A aggregating infromation from its neighborhood [3]

The described process is called neighborhood aggregation. If we want to predict node v, we need information from its neighborhood, meaning we need a way to propagate the message. Messages are passed and transformed through edges. All received messages are aggregated into a new message, via an aggregate function (usually sum), and then passed on. This is done systematically for every node in the graph [3].

The aggregation itself is done by a neural network. This implies the key difference from a typical ANN. Every node gets to define its own neural network and GNN is defined by multiple neural networks.

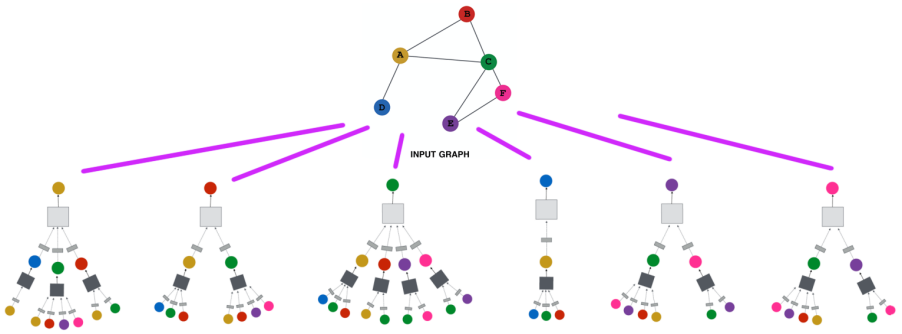


Figure 1.9: Neural network of each node for given input node graph [3]

Using a neural network for each node in the graph, we generate a low-dimensional vector representation, embedding. The network optimizes its parameters to capture important information about the node graphs. The optimization is typically done by minimizing a loss function expressing dissimilarity between predicted and targeted node embeddings. Similar nodes are close to each other, whereas dissimilar are embedded far apart [4].

At last, the embedding is taken to the prediction layer, where we apply a neural network curated for a specific prediction task. In summary, the GNN basically only transforms and extracts key information from the initial input graph to a reasonable scale and is only complete once we specify the prediction layer [4].

1.4.2.2 Prediction tasks

There are three general types of prediction tasks that GNN can do: graph-level, node-level, and edge-level prediction [4].

In **graph-level** task, the goal is to predict the property of an entire graph. The best example would be predicting molecule properties, such as the smell or what kind of chemical it is. It is the most basic prediction task, similar to image classification, where we predict a label or property to the input structure [4].

Node-level prediction predicts a property of a single node based on the graph connectivity. Zach's karate club problem is the best example of this type of task. The goal is to predict the loyalty of the club members between John A and Mr. Hi, who are having a feud. The graph represents individual allegiances. The prediction result is depicted in Figure 1.10. The task can be compared to image segmentation, where we would predict the role of each pixel in an image [4].

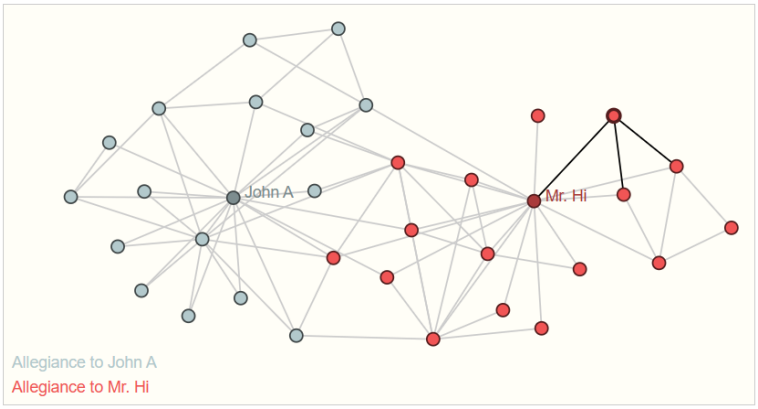


Figure 1.10: Zach's karate club allegiance prediction [4]

Last to explore are **edge-level** tasks. They are often used to predict and identify relations between objects. The input is often a fully connected graph, where each node represents an object. Looking at Figure 1.11, the task goes beyond simple image segmentation. It models relationships between all presented objects [4].

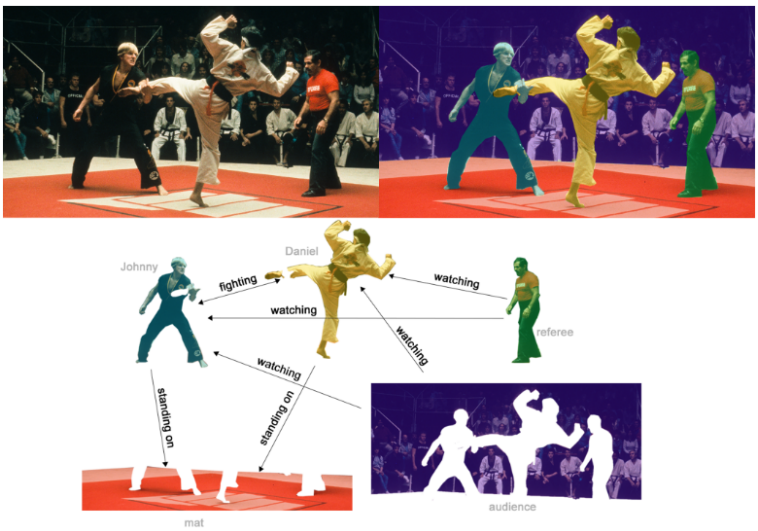


Figure 1.11: Image segmentation with predicted relationships between objects [4]

1.4.3 Recurrent Neural Networks

Recurrent Neural Network (RNN) is distinguished by its memory, which can handle input sequences of any length. Its past predictions influence the current

output. Resulting in different predictions depending on previous inputs in the sequence [35].

RNNs are widely used in fields like speech recognition, image captioning, natural language processing, and language translation and are found in popular applications such as Siri, Google Translate, and Google Voice search [36].

To illustrate how RNNs use previous inputs, consider the idiom „feeling under the weather“. To understand the meaning, the words must be in a specific order. RNNs take into account the order of the words and use the information from each word to predict the next word in the sequence. Each time step represents a single word, so for example, the third time step represents „the“. The hidden state of the RNN holds information from previous inputs, such as „feeling“ and „under“ [36].

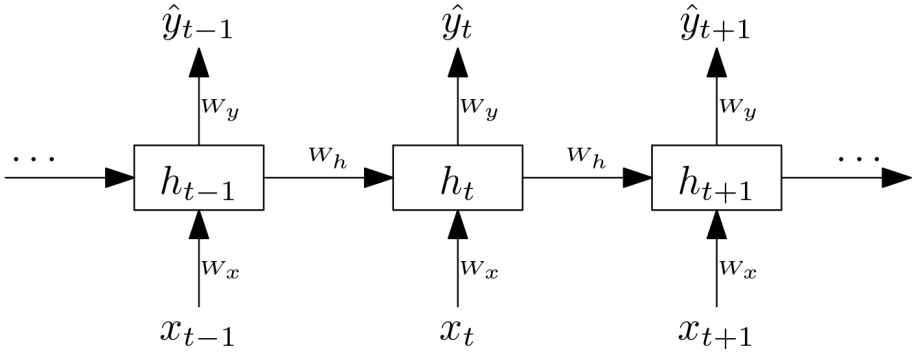


Figure 1.12: Unrolled structure of RNN [1]

Figure 1.12 depicts how the RNN network operates at each time step t. The current input $\vec{x_t}$ is processed by the network to produce the output $\hat{y}_t$, the next timestep of the input is x_{t+1} with additional input from the previous time step from the hidden state h_t. This allows the neural network to have a context from previous inputs while processing the current input. The ability to hold past values and work with a context in the data is due to the recurrent units, reffered to as memory [37].

The recurrent unit is calculated as follows:

$$h_t = f(W_x x_t + W_h h_{t-1} + \vec{b_h}) \tag{1.10}$$

f is the activation function, W_x, W_h are weight matrixes, x_t is the input, and $\vec{b_h}$ is the vector of bias parameters. The hiddent stat h_t at time step $t = 0$ is initialized to $(0, 0, ..., 0)$. The output $\hat{y}_t$ is then calculated as:

$$\hat{y}_t = g(W_y h_t + \vec{b_y}) \tag{1.11}$$

g is also an activation function, typically softmax, ensuring the output is in the desired class range. W_y is the weight matrix, and $\vec{b_y}$ is a vector of biases determined during the learning process.

Training RNNs uses a modified version of the backpropagation algorithm called *backpropagation through time* (BPTT). This process works by unrolling the RNN [31], computing the losses across each time step, and then using the backpropagation algorithm to update the weights. More on RNN in [28] by Lipton et al.

1.4.4 Long Short-Term Memory

Consider a task where we attempt to predict the last word in a sentence „The clouds are in the *sky*“. It is fairly obvious the last word is meant to be „*sky*“. The gap between the relevant information and the prediction place is small, RNNs can learn to use past information and make accurate predictions. However, if we consider „I grew up in Spain... I speak fluent *Spanish*“, the gap between the relevant information and predicting word is large. As the gap grows, RNNs are unable to handle the task. Such problem is called *long-term dependencies* [38].

Long Short Term Memory networks (LSTM), first introduced by Hochreiter S. and Schmidhuber J. [39], are RNN architecture with the ability to handle long-term dependencies. LSTMs replace RNN's hidden states with **LSTM Cells** and add connections between cells, known as *cell states* or c_t. Each LSTM Cell consists of three gates that regulate the input and output of the cell and its calculation runs as follows:

1. **Forget Gate**: Controls which information to keep and which to discard. *Sigmoid function* produces a value ranging from 0 to 1 base on the information from the previous hidden state and from the current input. The value closer to 0 indicates that the information should be discarded, while a value closer to 1 means it should be kept.

$$f_t = \sigma(W_{x_f} x_t + W_{h_f} h_{t-1} + \vec{b_f}) \tag{1.12}$$

2. **Input Gate**: Decides which information should be updated. The sigmoid function has a value between 0 and 1 based on the previous hidden state and the current input state. A value close to 0 indicates unimportant information, while a value close to 1 indicates important information.

$$i_t = \sigma(W_{x_i} x_t + W_{h_i} h_{t-1} + \vec{b_i}) \tag{1.13}$$

The information from the previous hidden state and the current input state are processed by a *tanh* function, which results in values ranging from -1 to 1.

$$g_t = \tanh(W_{x_g} x_t + W_{h_g} h_{t-1} + \vec{b_g}) \tag{1.14}$$

The decision on how to update the cell is obtained by multiplying sigmoid output and tanh output. With all the required values available, we can now calculate the *cell state* as follows:

$$c_t = i_t \odot g_t + f_t \odot c_{t-1} \tag{1.15}$$

3. **Output Gate**: Determines what information should the next hidden state contain. The previous hidden state and the current input are passed into a sigmoid function.

$$o_t = \sigma(W_{x_o} x_t + W_{h_o} h_{t-1} + \vec{b_o}) \tag{1.16}$$

Passing the newly modified cell state into a tanh function and multiplying its output with the sigmoid output, we get the hidden state [40].

$$h_t = o_t \odot tanh(c_t) \tag{1.17}$$

The output $\hat{y}_t$ is calculated the same way as regular RNN [1].

$$\hat{y}_t = g(W_y h_t + \vec{b_y}) \tag{1.18}$$

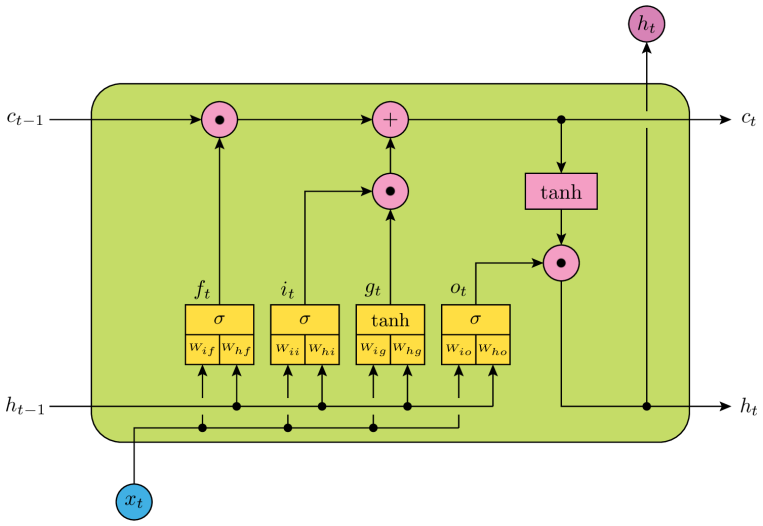


Figure 1.13: LSTM cell [5]

Chapter 2

Polygon Mesh

In the following chapter, we will discuss shape representation in computer graphics and explore some of their properties. We will also take a look at what tools can be used to optimize them in relation to mesh quality. Some of the usual data structures to represent a geometric shape and position within a 3D space are:

- **Mesh** is a set of vertices, edges, and faces defining a 3D object. It is especially used in computer graphics, and it is a simple way to represent complex 3D shapes. A face is a polygon with a minimum of 3 vertices. The most commonly used is a triangle. If a face is made of 4 vertices, it is called a quad, and more than four is called a general polygon. In our case, when we mention faces, we will have triangles in mind. These faces then form a general surface [41].

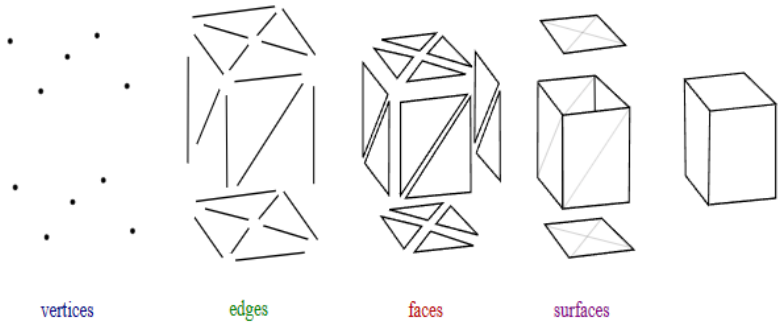


Figure 2.1: Elements of mesh object [6]

A 3D object can have a color. To do so, we assign color values to every vertex of the 3D object. The pixel color of the triangle is determined based on the three vertices by which it is made.

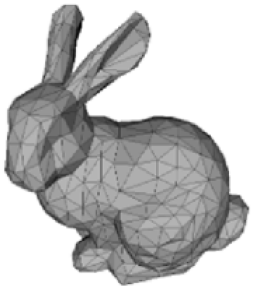


Figure 2.2: Stanford bunny model made of mesh [6]

- **Voxel** objects are in comparison with mesh objects solid. As already said, mesh objects are created from a surface of little triangles, but it is also worth noting that this object is hollow, just like a ballooon. On the other hand, if a model is created from voxels, abbreviation from volumetric pixels, it means that the object was created from cubes and the object itself is solid, its inside also holds information. The working scene is a 3D grid, and its data point holds information about opacity, color, and material information is often also stored [42].

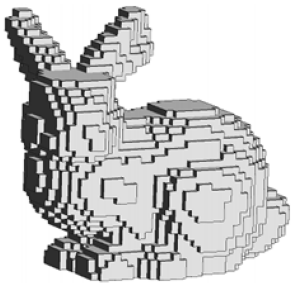


Figure 2.3: Stanford bunny model made of voxels [6]

  Voxels are often used in medicine and terrain representation. Voxel terrain can represent overhangs, caves, arches, and other, which is difficult to represent using heightmaps, which represents only top-layer data, and anything below it would be filled with no option for holes [42].

  The main disadvantage of voxels is the resolution. If we want to have a highly detailed voxel model, we would have to increase the resolution of the whole scene.

- **Point cloud** is a collection of points plotted in 3D space. Each point contains position coordinates, color values, and luminance values, determining how bright a point is.

Points are usually acquired by a 3D scanner or photogrammetry software. Scanners work by sending out pulses of light to the object and measuring how long each point takes to reflect back and hit the scanner. These measurements are used to determine the exact positions of points on the object, creating a point cloud. Photogrammetry is a process to create measurements from pictures. It uses photos of an object to triangulate points on the object and plot these points to 3D points, resulting in point clouds [43].

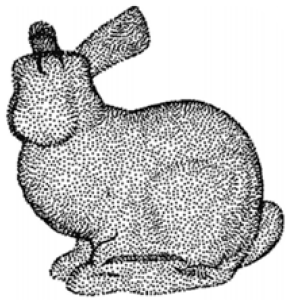


Figure 2.4: Stanford bunny model made of point clouds [6]

In terms of the thesis and its goal, we will mainly explore mesh polygons and their edge flow properties while also exploring polygon reduction operations on them.

2.1 Edge flow

Edge flow is a fundamental concept in 3D modeling. The general goal is to ensure that mesh edges follow the curves of an object. A mesh with distinctly different edge flows can represent the same object while preserving the same shape. The key difference and significance of edge flows plays in the world of animation, overall any deformation operation performed on the object. In general, a good edge flow has uniformly distributed points along the 3D model, meaning the length and the area of each primitive is also of similar if not equal sizes.

An optimal edge flow allows smooth and natural deformations. In the case of figure 2.5, if we want to animate various facial expressions, the left example would have distorted wrinkles. In contrast, the right example would allow for natural mimicry, as we are familiar with in real life. A more expressive example would be in 2.6, where we can see a plane of two different topologies having different behavior if bent. The left example has slight bumpy artifacts, while the right preserves smooth surface transition.

Edge flows are manually crafted by 3D artist and require experience to reach their optimal form. They are either being considered from the beginning

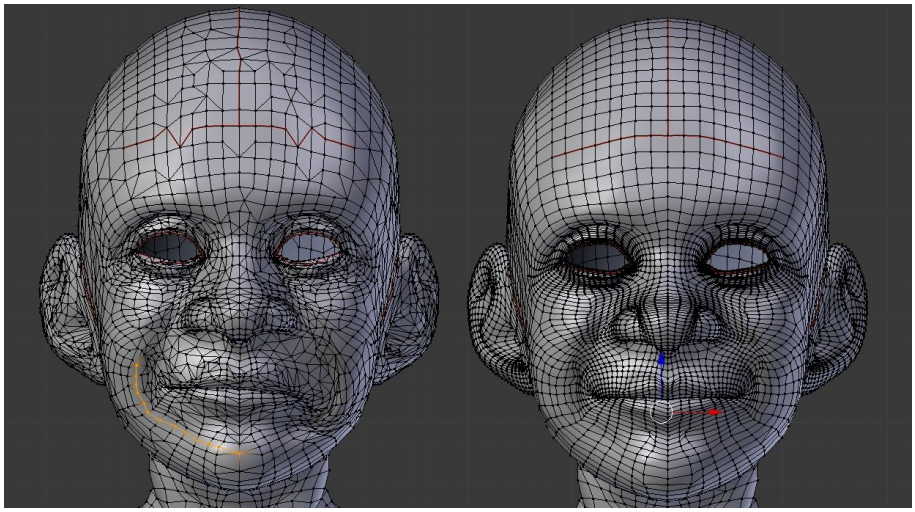


Figure 2.5: A mesh object constructed with 2 different edge flows [7]

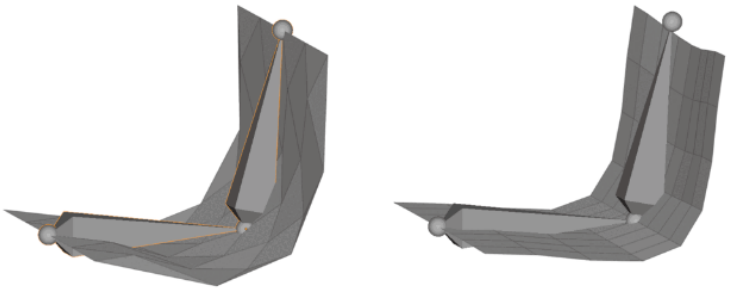


Figure 2.6: Deformation of the same object with different edge flow [8]

while creating the 3D mesh or as a post-correction, where the acquired 3D mesh does not meet the required quality needed for future work. One of the possible reasons for having insufficient quality could be caused by polygon reduction.

2.2 Polygon reduction

Polygon reduction is a common practice for performance and memory optimization as a higher number of polygons may introduce greater details, but they also bring high demand on hardware requirements. A high polygon mesh is usually a byproduct of the 3D scanning process where the resulting 3D mesh can be of enormous memory size. Most importantly, the reduced amount should be considered as the reduction can lose the visual representation of the object and its key features. The general goal is to balance visual

quality and performance optimization. Following, we will list some of the most used techniques for polygon reduction and their effect on the resulting edge flow.

2.3 Decimation

Decimation is a commonly used method for quick polygon reduction. It reduces the percentage of vertices and edges uniformly while preserving the overall shape of the reduced object, but it does not handle fine details well. Most 3D modeling software, such as Blender, Maya, 3ds Max, Zbrush, and Cinema4D, support decimation since it is a common technique. The overall control or workflow may differ, but the general concept remains the same [7].

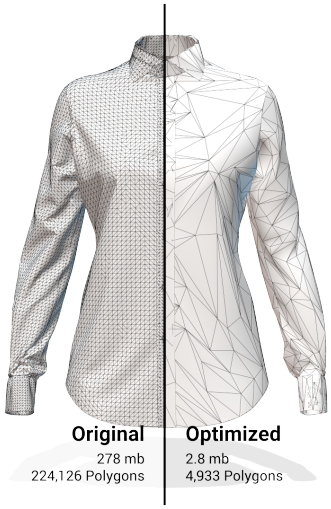


Figure 2.7: Decimation applied on shirt 3D mesh [7]

As seen in 2.7, the optimized 3D mesh may be optimized in the number of polygons, but its edge flow suffered greatly. If the shirt was part of any animated object, its part would deform unnaturally in undesired ways.

2.4 Retopology

Retopology is a semi-automatic method requiring high user input. It requires a user to manually outline the required surface by hand while hinting at its edge flow by following the model's edges before the 3D software follows the outlines and draws a polygon. This gives a user high control over its model but is highly demanding on general experience and skill while identifying the edges. From a quality perspective, the resulting model will be suitable for animations and smooth 3D model deformations. Still, from a workflow efficiency standpoint, producing one of these reduced models requires a lot of time.

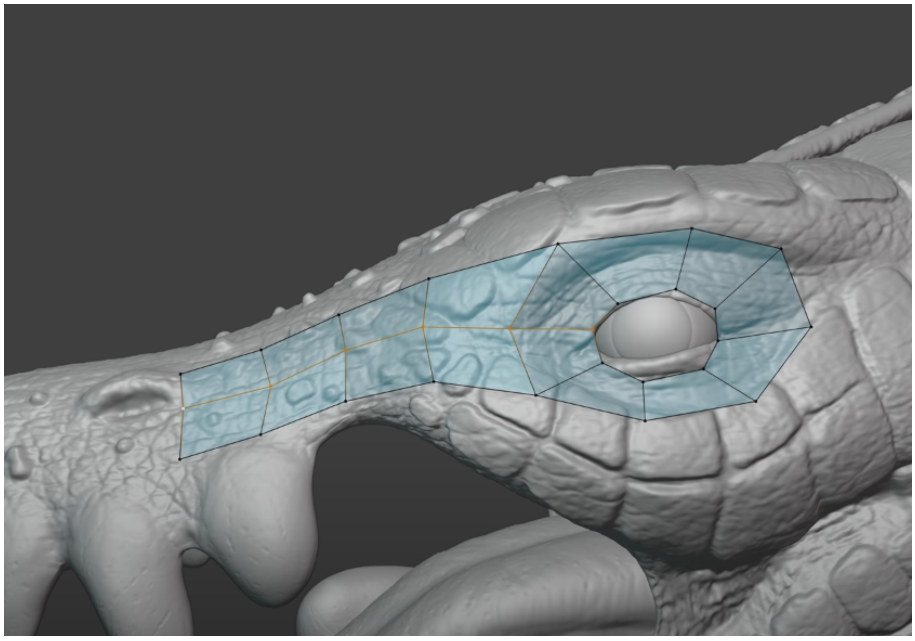


Figure 2.8: Retopology surface manually created lizard 3D mesh [9]

2.5 Instant Remesh

Instant remesh is considered to be current state of the art solution. The algorithm is currently implemented in Modo, but its source code with fully functional retopology application can be acces via GitHub. Instant remesh is leading in speed performance as well as in quality compared to other retopology solutions [44].

Most of other solutions using quad faces relying on idea base of finding out how each quad face should be oriented. The orientation can be visualized as little crosses covering the surface with flow lines overlay shown on top, seen in Figure 2.9.

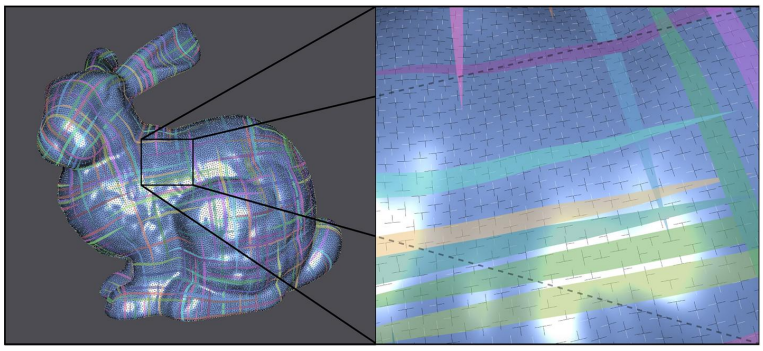


Figure 2.9: Quad orientation crosses visualized on rabbit mesh [10]

To reach the best remesh quality, we would want the crosses to be oriented in the same direction. Once the orientation is already known the algorithm then begins to place polygons.

Instant remesh reframed these problems into a special kind of smoothing operation allowing for a very efficient implementation. The algorithm subdivides the mesh iteratively into smaller batches and repeatedly blurs the data on these simplified meshes. The advantage in this approach is that all of the operation runs in parallel and is able to process meshes of hundreds to millions of elements [10, 44].

Instant remesh has also a unique approach of dealing with curvature, without the necessity of specifying any constraints, the remesher can automaticcaly align to corners and other sharp features [10, 44].

Chapter 3

Related Work

The main challenge is that there is currently no way to express the optimality of an edge flow by one number, which is critical for learning methods when we want to compute a loss function. We previously mentioned that an optimal flow has uniformly distributed points, edge lengths, and overall area of a primitive. One way to possibly achieve that is to look at the task at hand from a different perspective. Instead of doing a correction of the given 3D mesh, we will try to reconstruct the shape of a given 3D mesh with better topology.

In this chapter, we will explore research using neural networks for 3D mesh reconstruction.

3.1 Retrieve and deform a template

Retrieving and deforming a template is a two-step solution. First, a most suitable template is picked, and then it is deformed to the target object. The input is processed with a neural network that classifies which template is best suited, such as BCNet [11] or ShapeFlow [12]. BCNet method does not serve as complete shape reconstruction as more of a shape modification. The task was specific to body shape and cloth deformation. A most suitable body template was picked and then deformed to the target pose, which was conditioned by a single image input. An overview of BCNet architecture can be seen in Figure 3.1. Features are extracted from an input image extracted using CNN and passed to Multilayer perceptron to classify the most suitable template. The template is then passed on to deformation blocks, a combination of the Displacement network and Skin Weight network, which modifies the template to the desired pose with proper cloth deformation presented in the input image [45].

ShapeFlow, on the other hand, was more diverse in its shape templates, and the general purpose was to reconstruct a target object. A nearest-neighbor template was retrieved from the embedded space, and then a deformation

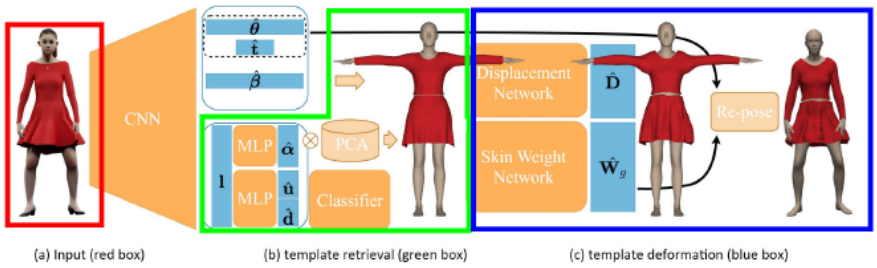


Figure 3.1: Overview of retrieve and deform shape reconstruction using BCNet [11]

network was applied. An overview of BCNet architecture can be seen in Figure 3.2. An input can be either a sparse point cloud or a depth map converted into a point cloud. ShapeFlow then learns key embeddings of geometric shapes based on deformation distances, meaning the ShapeFlow will choose the closest 3D template with minimal deformation required to reach the target shape [45].

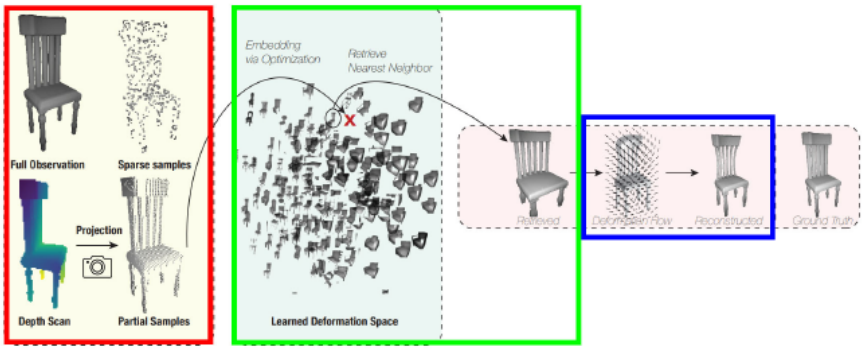


Figure 3.2: Overview of retrieve and deform shape reconstruction using ShapeFlow [12]

The results look promising with good quality. The limitation of the method is in its prerequisites, as retrieving and deforming solutions expect high–quality templates, which are also highly specialized in certain category types, which makes the overall input and output highly specialized. This defeats a purpose in terms of generalized solutions but rather is highly dependent on what templates are available in the domain.

3.2 Deform a primitive

Another method often used for reconstruction to target an object from a single image would be deforming a primitive. Picking a single primitive template that

would get deformed completely to the target mesh. Popular representation in Pixel2Mesh [13], Pixel2Mesh++ [46], and Neural mesh flow [14] was a simple sphere.

Pixel2Mesh had a 2D CNN extracting features from a single image, which was then leveraged by a deformation block, progressively deforming a sphere into the desired 3D model. The cascaded mesh deformation network is a graph-based convolution network containing three deformation blocks intersected by two graph unpooling layers. The architecture overview can be seen in Figure 3.3

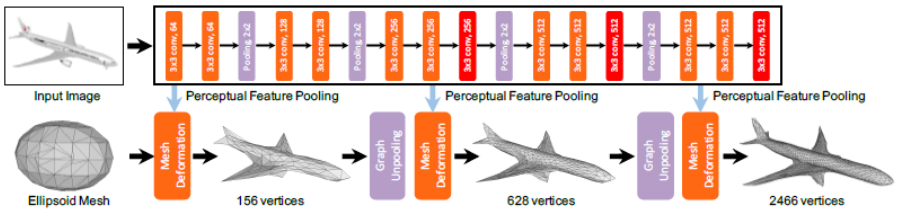


Figure 3.3: Pixel2Mesh architecture overview [13]

Graph unpooling block increases the number of vertices. With given face the points were added to the middle of each connecting edges, as shown in Figure 3.4.

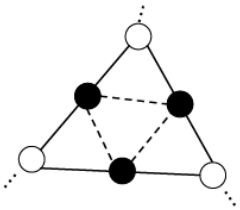


Figure 3.4: Graph unpooling [13]

Connecting these newly added points then forms a new set of primitive faces, while also ensuring even distribution of vertices and their degrees, making the mesh edge–flow stable.

We can see from the results how the Pixel2Mesh has difficulties with details of the meshes and some non-convex aspects of the target, such as wholes or other curve-ins.

Another work worth mentioning is Neural mesh flow, which explored mesh manifoldness, an attribute required for 3D printing and simulations so that they could interact with other real-world objects. The goal was to reconstruct a mesh accurately while being manifold [45].

Neural Mesh Flow (NFM) is a shape auto-encoder consisting of several Neural Ordinary Differential Equation (NODE) blocks that learn accurate mesh geometry by progressively deforming a spherical mesh [14, 45].

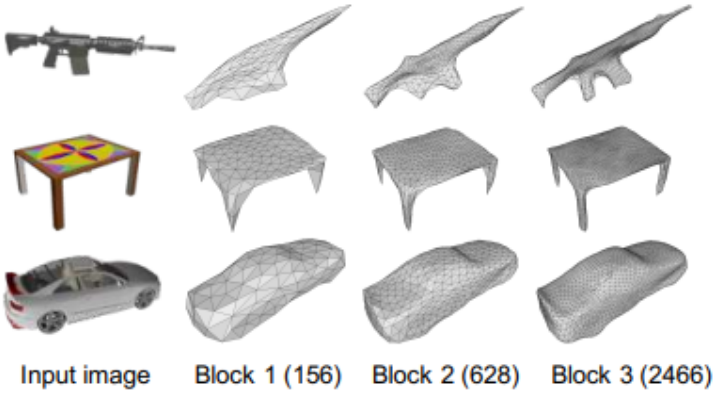


Figure 3.5: Prediction results of Pixel2Mesh [13]

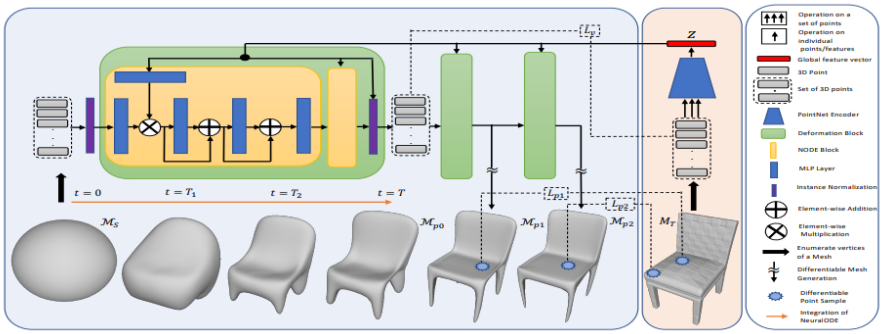


Figure 3.6: Neural Mesh flow architecture overview [14]

We can see the architecture overview in Figure 3.6. NMF broadly consists of four components. First, the target shape is encoded by uniformly sampling N points from its surface and feeding them to a PointNet [47] encoder to get the global shape embedding. Second, NODE blocks progressively deform the vertices of the template sphere towards the target shape conditioned on shape embedding. Third, the instance normalization layer performs non-uniform scaling of NODE output to ease cross-category training. Finally, refinement deformation provides a gradual improvement in quality [45].

We can see from the result in Figure 3.7 similar issues as in Pixel2Mesh, where the prediction has a problem with non–convex chair parts, the slight bend on top. Other than that, the general shape prediction is very promising.

Overall, the general limitation of the deform primitive method is how the primitive cannot be deformed to all 3D objects. Torus is one of the many examples where by deforming we cannot fully replicate the target torus without breaking and creating new connections for the sphere. Other more complex shapes, like cars with car seats, can be nearly impossible to get from deforming a sphere. We can create the outer shell of the car, but the interior seating can-

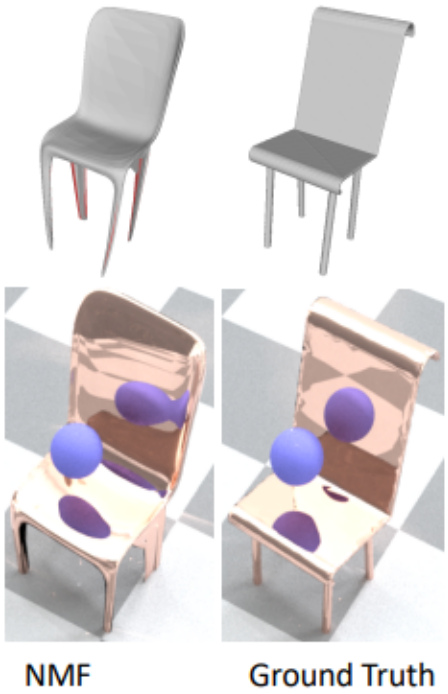


Figure 3.7: Prediction results of Neural Mesh Flow [14]

not be reconstructed. The general rule is we cannot successfully reconstruct shapes with holes or shapes with multiple layers of shape complexity.

3.3 Generative approach

The last one we will mention is the generative approach utilizing generative adversarial networks. The current state of the art is considered to be PolyGen [48], a generative model of 3D meshes predicts mesh vertices and faces, thus modeling a mesh directly. The architecture consists of two parts: an unconditional vertex model, which models mesh vertices, and a conditional face model, producing mesh faces conditioned by previously produced vertices [48].

In order to maintain sample diversity, the authors used nucleus sampling, which resulted in an effective way to reduce degraded samples.

PolyGen serves as a decoder, generating a 3D mesh and creating task flexibility where others only need to provide their task-specific encoder. This is different from retrieve-deform and deform primitive approaches, as they can be considered end-to-end solutions, that do not need any additional work but also make them less flexible. As seen in Figure 3.8, authors used image encoder, where in Figure 3.9 they use voxel encoder. The encoders result was then applied to the PolyGen decoder to generate the final mesh [48].

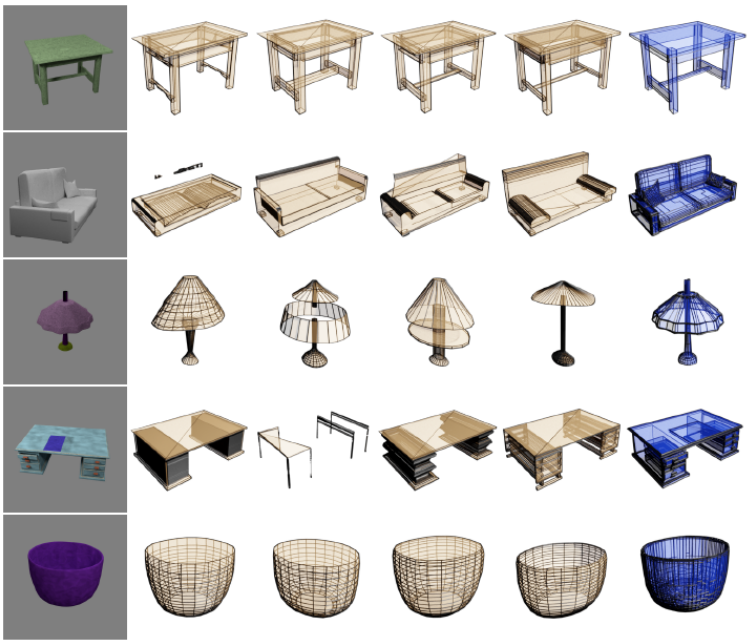


Figure 3.8: Image conditional samples (yellow) generated using nucleus sampling with top-p=0.9 and ground truth meshes (blue) [14]

The results are very promising, we can only notice a general flow in predicting more complex meshes such as the desk with drawers, where each drawer is modeled as an individual mesh. Also, these limitations can come from the encoders rather than the PolyGen decoder itself. The quality of mesh generation is highly dependent on the quality of encoders, whether they are descriptive enough to capture key features of the target.

3.4 Chamfer distance

Chamfer distance is used across all research to compute the similarity or often called likelyhood of two 3D meshes, hence used as a loss function for training. The distance is intended for point clouds. Thus, the 3D mesh surface is sampled to point clouds with a predetermined sample volume. It is known that when using chamfer distance as a loss function, meaning we are trying to minimize its output, the function can, at times, get stuck at a local minimum, and thus, it is common practice to apply regulations. Most commonly, edge, normal, and laplacian regulation, while also assigning weight to each regulation component.

Given two point clouds, X and Y, the Chamfer distance is defined as the sum of the distances from each point in X to its nearest neighbor in Y, plus the sum of each point in Y to its nearest neighbor in X.

Figure 3.9: Voxel conditional (blue, left) samples generated using nucleus sampling with top-p=0.9 (yellow) and ground truth meshes (blue, right) [14]

$$CD(X,Y) = \frac{1}{|X|}\sum_{x\in X} min_{y\in Y}||x-y||_2^2 + \frac{1}{|Y|}\sum_{y\in Y} min_{x\in X}||x-y||_2^2 \qquad (3.1)$$

It is base on greedy approach of nearest neighbor, which makes it difficult to use for non–convex shapes in reconstruction tasks, more specifically with while deforming a primitive.

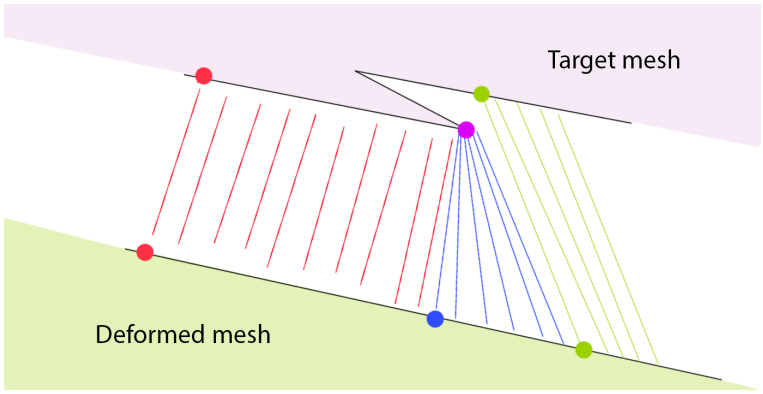


Figure 3.10: Computing Chamfer distance between target mesh and deformed mesh

Consider a simplified model situation depicted in Figure 3.10. We would like to compute the Chamfer distance between the target mesh and the deformed mesh. The computation progressively evaluates points on the mesh surface and evaluates the mesh similarity based on the closest opposite point. We can see no issue coming from the red point towards the blue point. Each point on the surface was correctly matched to its counterpart. The real problem comes from the blue point going forward. Instead of being matched to the inside of the non-convex corner, several surface points get matched to the purple point because, numerically speaking, it is evaluated as the closest. This would continue until the computation does not evaluate that the distance to the green point is closer and continues following the surface as expected. This is caused by Chamfer's greedy computational approach.

What we're trying to point out is that as long as we keep using Chamfer distance as our loss function, the reconstruction results will always have inaccurate artifacts, no matter how good the architecture may be, because the Chamfer distance will always neglect non-convex area in favor of matching the surface point with the closest counterpart.

Chapter 4

Implementation

The mesh reconstruction is often done from voxels, point clouds, single images, or multi-view images, respectively. There was previously no real desire in doing a reconstruction from 3D mesh as it does not give any additional value to the output mesh.

As previously mentioned, there is no way to numerically evaluate the edge flow's optimality, which poses a huge obstacle in learning methods. The base of any learning method is calculating a loss function between the ground truth and the predicted value. In our case, we wouldn't have such value to do so.

We tried to look at the problem at hand from a different perspective. Instead of trying to correct the given unoptimized 3D mesh, we would try to reconstruct it. The reconstruction would be done by deforming a spherical primitive, where it is guaranteed that the points and edges are evenly distributed. Our model would predict offsets of each vertex, by which we would have to move it in order to reach our desired shape. This of course gives constraints to what kind of meshes can we process. The limitation pointed our in Related Work are still relevant. Our argument is that we first want to explore the offset prediction for simple objects in this thesis. If the prediction is possible, we can then further explore complex structures. Possibly, in combination of the object segmentation field, where we could explore segmenting an object to primitives and then applying the deformation on each primitive, together combining to the initial target.

We could also argue why not use the PolyGen decoder, which allows us to generate complete 3D meshes and train our encoder, passing descriptive features to PolyGen. The main issue here is that we have no control over the quality of generated meshes, as simple reconstruction is not our only goal, but we would also want to achieve a certain quality of connectivity.

We propose an end-to-end solution taking a mesh input and reconstructing it by deforming a primitive of the same volume of points. The primitive is an equally point-distributed sphere, which gets generated at the beginning. This allows the model to be scalable to various volumes of points as opposed to most

reconstruction solutions, where the number of points is strictly fixed. We then apply a graph neural network on the input mesh to learn key shape features and then predict offsets of each individual point to deform the primitive to get the target shape.

4.1 Dataset

We had to be mindful when choosing which datasets do we pick for our training. We explored several promising mesh datasets, where we considered their suitability for primitive deformation. Our main criteria were to have diverse object categories while having simpler object shapes, avoiding holes and non-convexity. Many 3D shape datasets are too specific, eg. human scans in various positions, furniture, and scenes.

4.1.1 Shapenet

A Shapenet [15] dataset is widely used for 3D deep learning tasks and is one of the largest publicly available datasets with diverse object categories. The main flaws of the dataset are obscure normals and corrupt face data, making it difficult for other 3D frameworks to perform any operation on the given mesh. The authors had already addressed some of the issues with releasing the second version, but with the slight improvement in quality, the overall issue is still present. Another reason would be how many of the shapes were complex structures assembled from multiple mesh surfaces. Considering these things, we deemed a Shapenet dataset unsuitable for our use.

4.1.2 Tosca

According to the published documentation, the Tosca dataset contained many organic meshes of animals, mythical beasts, and human figures. As promising as that may sound, the provided links were all dead. Neither the official site, nor official PyTorch dataset loader built specifically for the Tosca dataset had updated working links to the resources.

4.1.3 Shrec

The Shrec dataset [17] contains various categories, from organic to non-organic, 19 categories in total. Each category has 20 samples. This suits our needs perfectly as we are able to cover categories in a more general manner. Samples are also represented by seamless surfaces, without defective holes or gaps – a key difference compared to many other collections. This is a great benefit as we are already limited by the non-convexity and structure simplicity. Seamless surfaces also make the dataset very suitable for primitive deformation.

Figure 4.1: Shapenet data samples [15]

4.2 Preprocessing

Compared to previous work related to 3D mesh reconstruction, the general flow was to compute the loss function via the Chamfer distance. Comparing the likeness of the shapes. This approach is limited by Chamfer distance's limitation of working with non-convex shapes, which poses a big challenge to the greedy approach of Chamfer distance. We covered this limitation briefly in the previous chapter.

Instead, we would compute the loss function by comparing the offsets, not having the target mesh as a ground truth but the necessary offset to reach the target mesh serving as a ground truth. The sphere primitive is required to have an equal number of points as the target mesh.

The glaring question is, how do we calculate the necessary offsets? Despite having an equal number of points, the calculation of the distances is more complex than it seems. The reason is the different indexing of points between our template primitive mesh and target mesh. We would need a deterministic reindexing solution that takes into account the connectivity and vertex degrees. We would want to find the isomorphism of two graphs, which is a very hard problem. Also, consider the domain in which we are working, where the number of points can be ten thousand. A simple brute force approach for each sample is computationally too expensive and almost impossible with given resources. We considered a solution by which we would choose either the highest or lowest point of the target mesh and the primitive we were about to deform. This point would serve as a reference point. We would then apply the

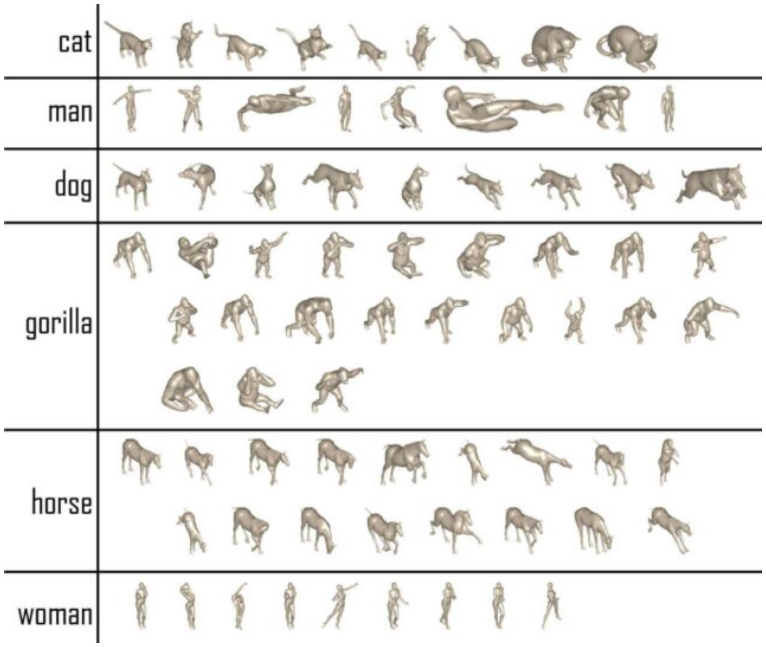


Figure 4.2: Tosca data samples [16]

breadth-first search (BFS) [49] algorithm on both meshes simultaneously, reindexing two meshes and expecting the same point indexing. This approach is not sustainable for meshes, which would have slightly different graph topology and edge connectivity. The BFS algorithms explore the point's neighborhood and schedule the search accordingly. Different topologies would cause the algorithm to search in different directions, making the reindexing still incorrect. We wouldn't be able to simply calculate the Euclidean distance of two points as that would require strict indexing similarity. Another solution we were considering was to combine a greedy approach with BFS. We would divide the mesh into point batches. In graph terminology, we would simplify the graph into subgraphs. Instead of running BFS on individual points, we would let the algorithm traverse the batches as they are neighboring each other. While visiting the batch, we would apply a greedy algorithm to find the closest point, similar to the Chamfer distance computation. Using BFS this way would allow us to explore non-convex areas systematically while not being bound to indexing because the distance would be utilizing Chamfer's distance greedy approach. These are hypothetical ideas, as implementing the proposed algorithm requires significant time and dedication, and the solution falls into the field of graph theory exploration.

We ended up having to utilize the Chamfer distance in a more stochastic manner. Deforming the primitive until it forms our target mesh. We do so by minimizing the Chamfer distance function. We then save the offsets as our ground truths for our training.

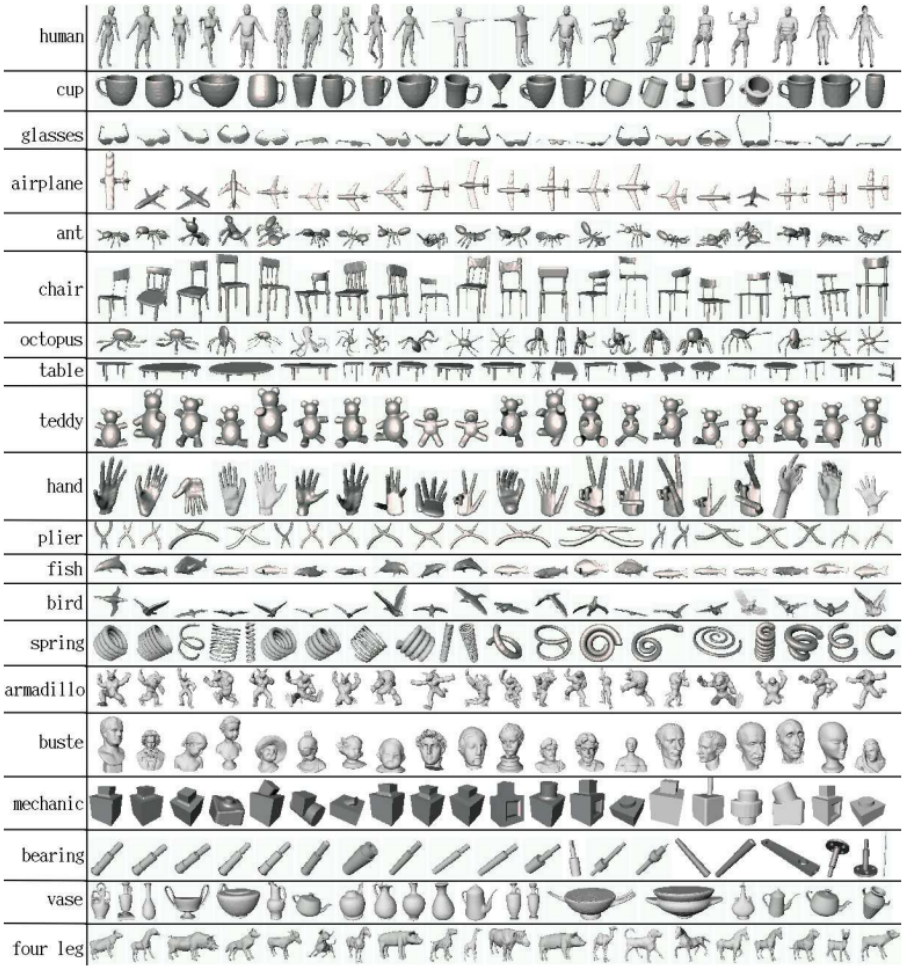


Figure 4.3: Shrec listed categories with mesh samples [17]

We had to consider a few things before starting computations. First, how many iterations do we want to run the minimization, and what would be the learning rate for our optimizer? The number of iterations should be reasonable across all meshes. The goal is to find a number by which the mesh would not be deforming anymore, meaning the Chamfer distance had found the global minimum or got stuck in the local one and could not get out of it. We could always choose a high number of iterations, but we must take into account the computational feasibility in terms of hardware availability and waiting time. We also don't want to cut off the computation too soon, as the mesh might not have gotten deformed to its best quality potential. Not to mention, each mesh has different computation demands – one can take longer than the other.

One option we considered was to stop the computation dynamically as needed. The expectation behind the approach was that we would save neces-

sary computation time where it's not needed and, on the other, provide time where there is still potential for improvement. We would have looked at the Chamfer distance value and calculated whether the value is still decreasing or not by comparing a difference between the current value and some previous value with a certain threshold.

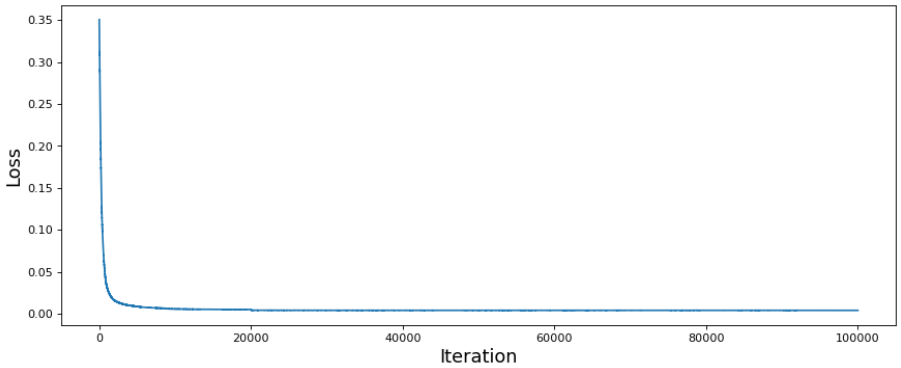


Figure 4.4: Chamfer distance computed on 10 000 iterations

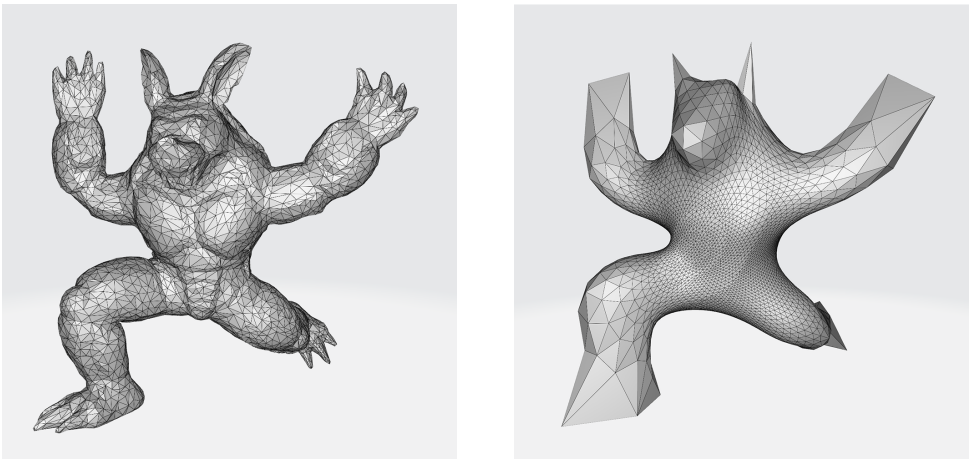
If we look at Figure 4.4, we would assume at first glance that the distance value is stuck and the mesh is not improving anymore, but if we explore the produced meshes, we will find out that it is not necessarily true and the mesh is still being improved. These improvements may seem insignificant as they do not necessarily translate to the plotted values, but in truth, they are the most important ones. This portion of computations tries to further improve non-convex areas or correct defects like holes or outlying points from the main shape, and it is highly desirable to let the computation to do so as much as possible. Also, the value can, at times, oscillate at one point for a certain time but then drops as the minimization was able to get out of the local minimum.

We ended up falling to the simplest solution. We looked at the most complex meshes and explored how they deform across the iterations and noticed when the deformation becomes insignificant. We will undeniably compromise in terms of unnecessary computation time for simpler meshes.

With all things considered, we decided to run the computations for 80 000 iterations. The first 50 000 iterations were for shape determination, and the last 20 000 iterations would be reserved for potential corrections.

The last thing to mention is the regulation components applied to the Chamfer distance. We previously mentioned that it is a common practice to apply regulation components to Chamfer distance as the distance itself is keen to fall to the local minimum if minimized. Each component also represents a certain property and affects the deformation result. Edge regulations make sure the length is approximately of the same scale. Normal regulations consider normal consistencies, so the mesh would not have obscurely flipped faces, whereas neighboring faces would have normals in opposite directions. Lapla-

cian ensures the mesh is smooth. We let edge regulation have full weight, as it is desirable for us to, in terms of edge flow, for the mesh to average the same value across all edge lengths, meaning we set the weight to be 1.0. Normal weights are important, but if given too much weight, our mesh would be held back from making sharp bends in its shape. Thus, we give it a lower priority and set the weight to 0.1. The only component left to discuss is laplacian regulation, in which we give weight 0.5 for initial 20 000 iterations and then lower its weight to 0.1. This way, we force the sphere to capture the target's general shape quickly, and then by lowering the constraint for smoothness, we allow the deformation to capture key shape characteristics of the target shape.

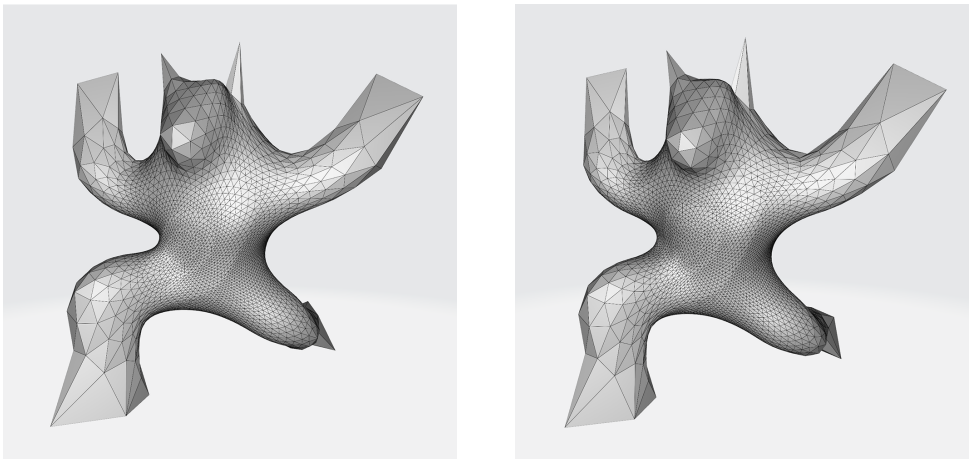


(a) target mesh (b) computation result

Figure 4.5: Comparison between target mesh and Chamfer distance minimization result

Looking at Figure 4.5, we can see the computation result being very poor. This makes our decision to use Chamfer distance for data preprocessing questionable. Not only did we discuss in the previous chapter how the output of previous research was limited by using Chamfer distance, which has difficulty working with non-convex shapes. We can now also visibly see how the pure minimization is very poor in the deformation/reconstruction task. Other than the general shape, we were not able to preserve fine details of the mesh, nor able to handle non-convex areas like fingers.

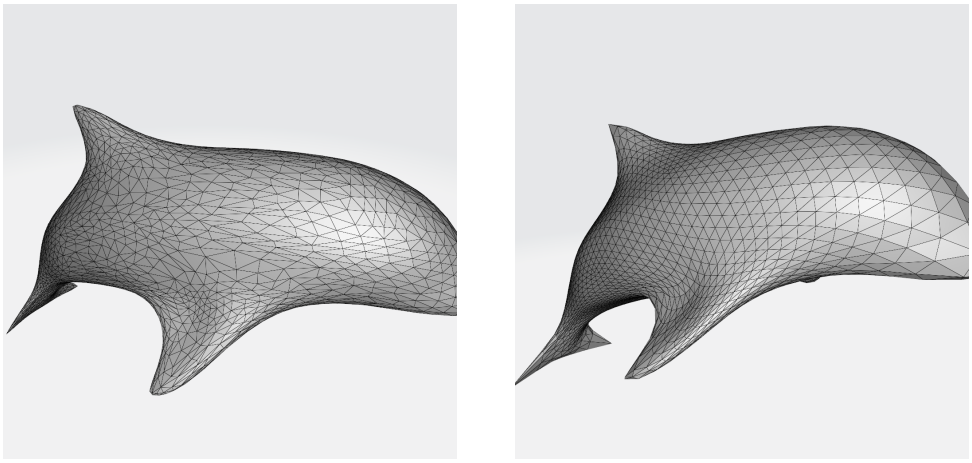
The mesh improvement speed, determined by the learning rate of the optimizer, also closely affects the overall quality of the computation result. Choosing 0.01 and greater makes the minimization speed very high, with the risk of overshooting the minimum as the minimization step would be too great ever to reach the desired value. Meaning the mesh reconstruction would not be able to reach its best potential. Conversely, if we explore the difference between 0.001 and 0.0001, we would see little quality difference, seen in 4.6. We



(a) learning rate = 0.001 (b) learning rate = 0.0001

Figure 4.6: Comparison between the result of 0.001 and 0.0001 learning rate

definitely cannot label the mesh as optimal, but it's shown that unoptimality comes from the Chamfer distance limitation itself. On the other hand, if we take a closer look, we can see hints of improved edge flow distribution in the deform result in Figure 4.7. The general distribution of points and length of edges are evenly distributed, thanks to regulations during the computation.



(a) target mesh (b) computation result

Figure 4.7: Closer edge flow comparison between target mesh and minimization result

We would like to remind the reader that the intention behind using Chamfer distance was only to compute the offset values and not base the model learning on the distance function itself. We had to resort to this method as

there was currently no feasible option to extract the offset values. Exploring better computation solutions could be part of a follow–up work, as this poses a challenge beyond the scope of this thesis. We could explore how to compute the offsets effectively and accurately. This way, most of the prediction flaws could be detached from the architecture quality and instead be blamed on the preprocessing stage, which could be improved further down the line. To rephrase, the goal is to remove the limitation of the Chamfer instance from model architecture quality and instead have the limitations affect the data quality.

4.3 Architecture

Regardless of what the actual architecture of the model looks like. Our pipeline first retrieves a primitive by generating a sphere mesh with the same number of points as an input mesh. The input mesh is then passed to the model to learn its shape features and predict offsets required to deform the primitive. These offsets are then applied to the primitive retrieved in the first stage, forming the desired target shape.

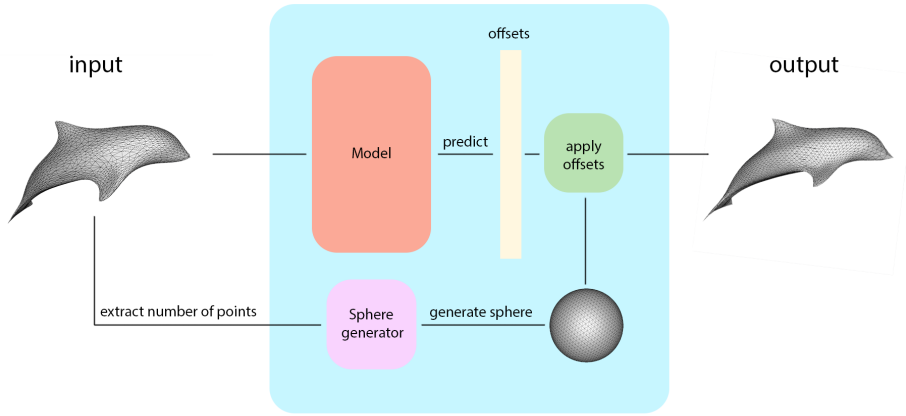


Figure 4.8: Overview of the prediction pipeline

We have already insinuated a few times that meshes can be represented as graphs, the analogy between vertexes and points, edges, and connections between points. Any mesh can be represented as a graph, where each vertex holds positional features. This brings us to the difference in terms of topology, where in graph terminology, the topology is not positional aware. It does not matter where the vertices are positioned as long as the graph connection is preserved with all its vertex degrees. With meshes, the spatial position in graphs is a key descriptor for the shape. Just as graphs, meshes do not have spatial locality in general. They are arbitrary in size in terms of the number of points and complex topological structures.

With that said, when choosing an appropriate architecture, we decided to explore graph neural networks. We'd use a graph neural network to aggregate mesh features and then apply linear regression to get the final offsets. Input features consist of point coordinates, and output is an offset coordinate by which the point should move. Both are 3 positional coordinates in Euclidean space. Both in real numbers are not being fixed to any domain. Thus, linear regression is the most suitable predictor.

We explored Pytorch Geometrics, a Python framework specifically implemented for graph neural networks. The library provides most of the state-of-the-art architecture, making them available for easy use and scalability while also providing lower–level operations.

Among many listed networks, we turned our attention to Residual Gated Graph ConvNets [50], Graph attentional networks [51], and Topology adaptive graph convolutional networks [52], as they were viewed to be most specific to our task at hand while also being most powerful from the listed implementations.

We looked at attention networks because they often outperform typical graph neural networks. When looking into the definition of the network, we find they are based on assigning weights to neighbors during the neighbor aggregation process, giving neighbors different priorities. In terms of meshes, it is not necessarily desirable. Each point holds its own significance in terms of the overall shape. Thus, we would end up in a situation where every neighbor would have the same weight, neglecting what makes the attention network unique.

Topology adaptive graph network considers graph topology in the learning process, which, by the sound of it, looks like the best option for our case. Unfortunately, this topology relates to graph topology, which disregards any positional awareness. As we previously mentioned in this chapter, the graph topology is not the same as the mesh topology. To further explain, consider our primitive mesh, being deformed only by offsetting it points, not changing any of the point connections. In terms of graphs, the deformed primitive has the same graph as the original primitive, thus its graph topology. In terms of mesh topology, the two are distinctly different due to the shape and how the edges flow through the shape.

After considering topology adaptive graph network and attention network, it led us to believe that we should use simple graph neural networks, but when exploring our data, we reached the conclusion that a simple graph network wouldn't be sufficient. If we look at the mesh data, we can see how points are related to each other to form the final shape and how points can get very far from each other (many points, neighbors, in between) depending on the number of points in the mesh. Thus, the passed information can get lost in the transition in the neighbor aggregation process, which would lead to long-term dependency problems. This is addressed by residual gated graph networks, which are based on recurrent networks and the LSTM concept, where the

LSTM cell is specifically made to tackle long-term dependency. Thus, we came to the conclusion to use the residual gated graph network as it was viewed to be the best option from the listed available architectures and the best from the narrowed-down list of potentially promising architectures.

4.4 Training

As to this point, we possess everything needed for training. We have a completely preprocessed dataset, which we split into 80% for the training set and the remaining 20% for the testing set. The data was scaled to fit in a sphere of radius 1 and centered to $(0, 0, 0)$ coordinate. We chose a residual grated graph network to make the core of our model, where the last output layer would be determined by linear regression. As for the network activation function, we used ReLu, and for the loss function, we used traditional mean squared loss. The input layer is of dimension 3, as well as the output layer, as the dimension correlates to the Euclidean coordinates. The remaining question is, what dimension should hidden layers have? Higher dimensions would mean greater training time, but also more information can get encoded for information passing, while lower dimensions would imply better training speed but with possible information loss.

We began with 256 to see how the training would perform. The dimension viewed to be quick enough to learn the features when we experimentally tried the network implementation. By that, we mean overfitting the network on one sample if it is able to predict the offsets. The network adapted fairly quickly to the sample and yielded offset results corresponding to the target shape. We must note that by this, we only tested the overall pipeline implementation but not the actual power of the model.

With all things in place, we applied the data to train the model. Unfortunately, the model did not improve from the initialization values. The loss function decreased only slightly before starting to converge. We can see the trend in Figure 4.9, where the values are not decreasing anymore beyond 400th epoch, while also not converging at satisfactory value.

First, suspicion was drawn to the training data, where we thought that some obscure samples caused by the preprocessing limitations could hinder the model's ability to learn the task at hand. The obscure samples could introduce too much noise for the model, as outlying points could lack any logical cause as to why they should be placed at the given position. They were causing the training to be confused and propagation from loss function to be incorrect.

We removed these samples, leaving only clean data, and the training was improved, as the loss function had now decreased more, but still not enough. The training began to converge again, only now slightly delayed at a lower value. We have tried various model changes, such as increasing and decreasing

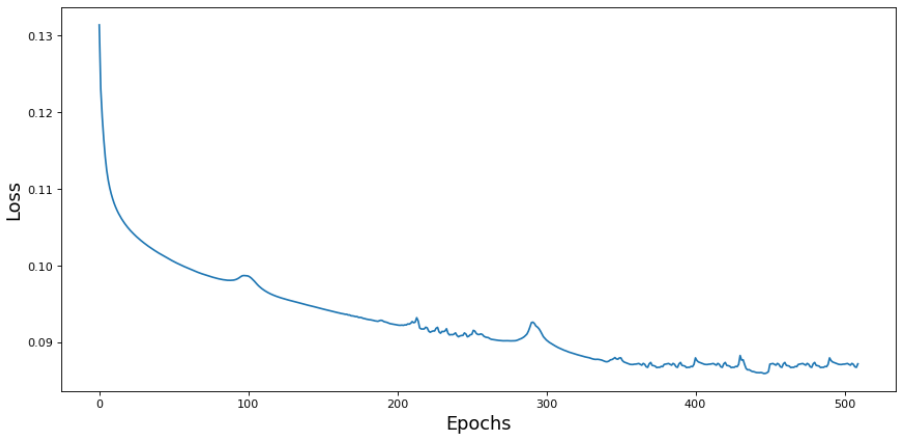
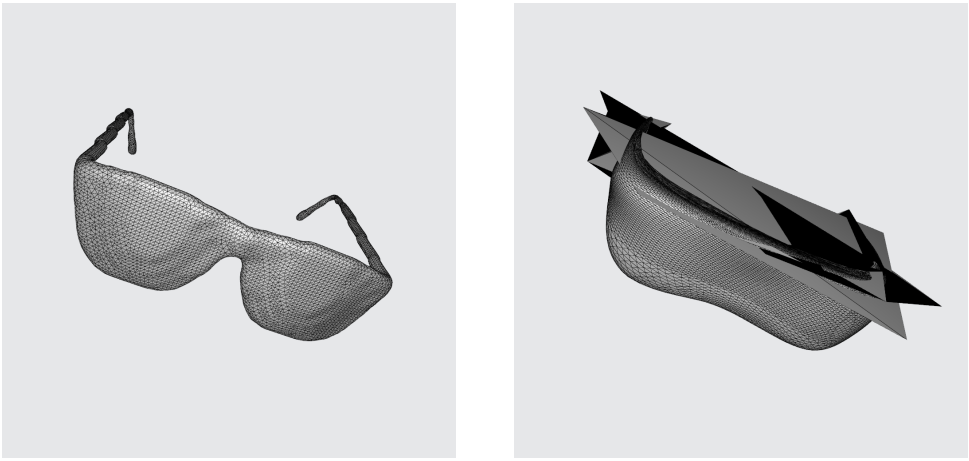


Figure 4.9: Loss function convergence



(a) target mesh (b) computation result

Figure 4.10: Comparing defected sample with its initial target sample

the number of hidden layers, but the only noticeable change was in training speed, but the converging problem still remained. Nor increasing the dimension of hidden features helped, as the loss values might been different, but the training defect seemed to persist. Both changes only affected the training speed in-terms of real-time spent on one epoch, other than that the general the general trend of loss function remained the same.

We also tried to have dimensions of hidden layers internally increase gradually and decrease gradually to avoid the possible shock of drastic dimensional change, but that only degraded the training loss. We thought whether having input coordinates centered at (0,0,0) could cause a dying ReLu problem, an issue where the activation function is unable to update the weights because too many negative values are present at the input. Thus, we changed the

ReLu activation function to Leaky ReLu, which directly addresses the dying ReLu problem, but the change had no effect on the overall performance.

We began exploring hyperparameters and our initial learning rate, which was 0.00001. If we increased the rate to 0.001, the loss value would spike up at a certain point, as the optimizing step was too big and had overshot the supposed minimal value. We decided to apply several machine learning techniques, like adding a dropout layer or batch normalization, which seemingly helped, as the loss value was able to pass the previous converging point and then proceeded to decrease very slowly. We could assume that the model is training very slowly, but as the number of required epochs was still increasing and the loss function still not converging, we were concerned by the model's learning ability. The general understanding is a number of epochs required to train a model should not reach such magnitude, as the high number only leads to overfitting as a result.

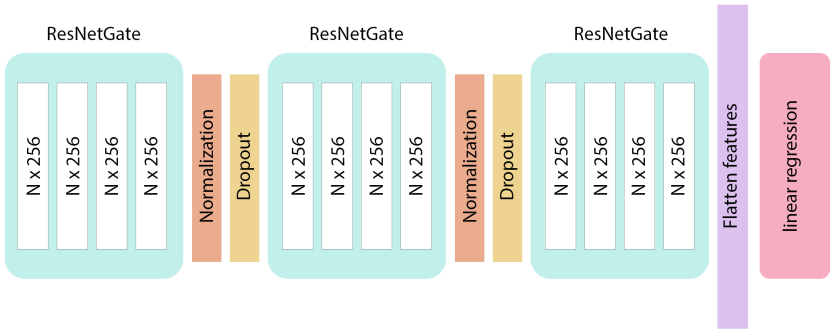


Figure 4.11: Final state of the network's architecture

After several changes like increasing the number of hidden layers to 12, to make the model more robust and applying normalization and dropouts, we reached the final form of our network, depicted in Figure 4.11. Normalization and dropouts are often used in deep learning to increase stability and prevent overfiting. We can see normalization and dropouts being applied after each of the 3 blocks of hidden layers except for the last block, where the features get flattened to a vector and applied to the linear regression. As for the optimizer's learning rate, we kept the initial value of 0.00001.

To investigate the strange learning behavior in regard to a large number of epochs required. We took this architecture and resorted back to the stage, where we forced the model to overfit on one sample. The overfitting happened faster than before, which was expected due to the improved model. We then proceeded to train on two samples and saw the overfitting requiring more epochs to happen. As we were adding on to samples, the required training time was only increasing, requiring more epochs to minimize the loss function. With only 20 samples, the loss function value quickly attained the strange trend we observed while training on a clean set. We can see in Figure 4.12,

that the loss value keeps decreasing and if kept increasing the number of epochs the loss would only decrease further.

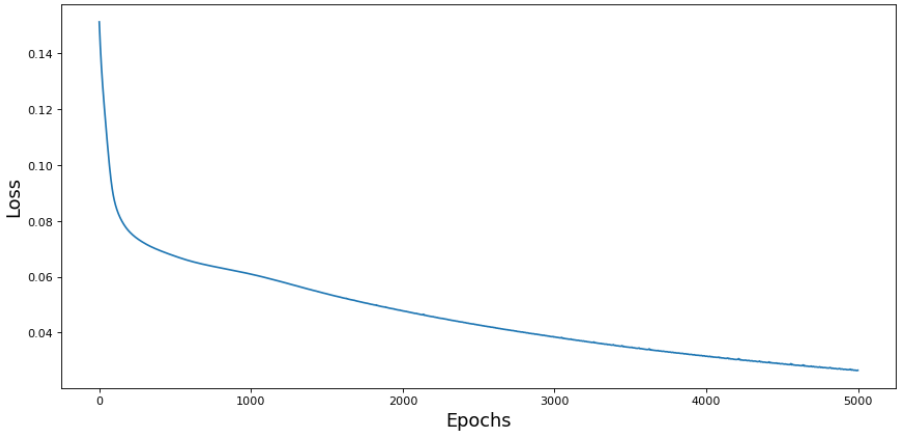


Figure 4.12: Loss function on 20 samples

This only indicates that the model was still not learning in the manner it should, learning the shape features and predicting based on them, but, in fact, was overfitting. The model was trying to match the training set to its ground truth offsets instead of learning how to predict them. This fact could also be proven by observing the loss value from the test set, where the value did not improve along with the training loss. All these trends would still be present if we were to change to other graph network variants, such as attention networks or topology adaptive networks, which is unsurprising considering previously made statements regarding these types of networks. The only difference was, again, the value at which the network began to behave strangely and the time needed to get to that point.

With this, we had to conclude that the model was insufficient and unable to learn the key shape features. We were not able to find the reason for the training disfunction. We could argue that the quality of the preprocessed data was not sufficient enough, the insufficiency is in terms of reconstruction quality. Each sample may not capture all the details of the input mesh, but they capture the general shape without the defect of the outlying points as they were removed from the training set. In other words, we could view the training task as stylizing the input sample to a smooth primitive shape. This should only affect the resulting prediction but not the overall learning process.

The final possible thought is that the node feature and point coordinates were not enough to describe the shape characteristics. The question is, how can this be improved? We tried to extend the feature vectors by normal coordinates, changing the input dimension from 3 to 6. In hope to make the features more expressive. We observed improvement in training speed. What previously took 5 000 epochs, now takes 500. The result is still unsatisfac-

tory as the loss value is still not at the point we need. This hints us to the possibility that in order to be successful, we would need to improve the input feature vector and provide more information expressing the shape as the point coordinate and its normal is not enough.

The last idea would be to change the representation from a point graph to a face graph, where each node would contain information about the normal coordinates and coordinates of the points forming this face. The problem with this approach was the question, what would the order of listed points and their winding be, and which point should be listed first? The face lacks any locality. Thus, we wouldn't be able to determine which point should be the starting one. We are afraid this would bring certain randomness to the data and would only confuse the model from learning the shape features.

In conclusion, we were not able to successfully train the architecture to perform offset predictions. We tried several optimization techniques and made several changes in the hope of improving the learning process, but aside from slight improvements, the end result was still the same. The network kept converging too soon and was unable to learn key shape features.

4.5 Code implementation

Despite the unsuccessful attempt to train the model, we fully implemented the pipeline depicted in Figure 4.8. We used the PyTorch framework with its additional Geometrics and 3D library made explicitly for graph neural networks and 3D object processing. PyTorch3D provided many useful tools and data structures for easy manipulation and processing of 3D meshes for neural network training, providing many methods to convert the data to tensors, base units used in neural network implementations. PyTorch Geometrics, as previously mentioned, provides many implementations of state-of-the-art graph neural network variants while also providing much lower-level operations.

Our implementation has 3 main entry point scripts: data processing, model training, and prediction. All are executable as is, provided the necessary required requisites, such as data to be processed, data for training, and model to load with an input mesh to be predicted. All entry points have CUDA acceleration options, enhancing the computational speed if NVIDIA CUDA is available for use.

Entry points share common utilization functions, such as IO operations, PyTorch3D wrappers, and data loaders implementation in the form of a shared library to save the code duplication and serve the purpose of unification. The code is documented and can be easily extendable if any investigation of why the training failed was to occur.

Chapter 5

New State of the Art

In time of writing this thesis a new research, InstantMesh: Efficient 3D Mesh Generation from a Single Image with Sparse-view Large Reconstruction Models, was published. Introducing new state of the art in 3D mesh reconstruction from a single image. We did not cover this publication in Related Work, because the work was published a week before finishing this thesis. Despite that, the results are undeniable, tackling various issues we mentioned in the thesis, but were unable to solve, most notably shape non-convexity and complex structures with holes and gaps in relation with data preprocessing. Thus, we would like to take a look at the InstantMesh solution in the following chapter and explore how the authors solved the reconstruction problem. Also, explore whether the findings are applicable to the edge flow optimization.

InstantMesh is very impressive in 3D assets generation in terms of quality and speed as one high quality asset is generated within 10 seconds.

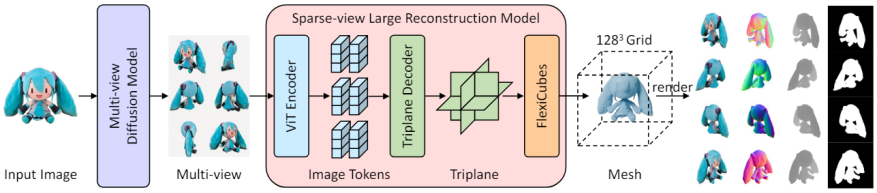


Figure 5.1: InstantMesh architecture overview [18]

We were most intrigued about whether the authors used Chamfer distance in their implementation and if so, whether they encountered the limitation we pointed out in the thesis. Further reading the publication showed authors using multi-view diffusion model. The diffusion model generated 3D consistent multi–view images from an input image which were then used for Large Reconstruction Model (LGM). The backbone of LGM is a transformer decoding image tokens to triplanes, but instead of requiring volume rendering to synthesize the result, InstantMesh uses FlexiCubes to rasterize the mesh.

This significantly improves memory efficiency [18].

In terms of loss function, authors did not use Chamfer distance but opted for image loss and mask loss while also comparing rendered depth images. This does not confirm our statements made in the thesis, but it also does not undermine it, as these results were not achieved with Chamfer distance involved.

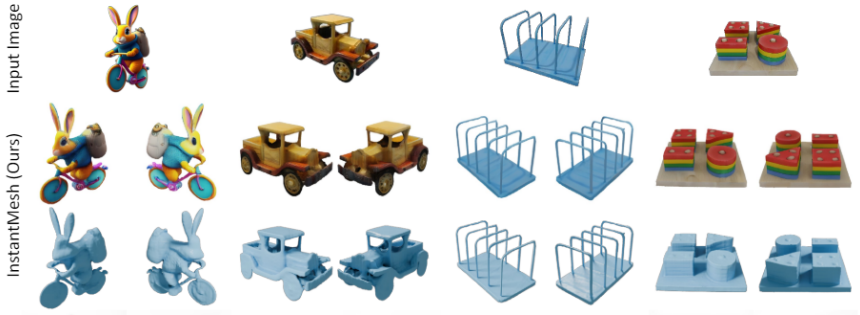


Figure 5.2: 3D meshes generated from a single image by InstantMesh [18]

Considering the generation out of a single image, the final generation is quite impressive without noticeable shape defects. FlexiCubes improved the results in terms of smoothness, but the authors mentioned limitations in modeling tiny and thin structures. Another limitation was the resolution in which the decoder was working. This could lead to significant limitations in high–definition 3D modeling. From our observation, we also noticed how the model generation is formed from one surface, not respecting individual parts of the object. Yet, we think this is beyond the scope of this research and could be explored further in the field of object segmentation [18].

Regarding edge flow optimization, the asset generation speed could possibly compete with analytical solutions, but the analytical approach would still be superior in preserving individual parts separated. Yet, we don't think this direct mesh prediction is much applicable to edge–flow optimization task as the architecture is still very focused on shape quality instead of mesh quality, meaning the architecture is trying to solve different task.

We could think of replacing the MultiView Diffusional Model and the encoder with GNN to aggregate shape features, but the generative part would need to be modified to focus more on the mesh quality, ending up completely changing the architecture and not really utilizing the findings by the publication. Providing well–optimized ground truth would not help because the loss function is based on shape quality without any point or edge awareness.

Chapter 6

Conclusion

We explored the current state of edge-flow optimization and tried to solve the issue with a stochastic approach, which had not been previously done. We encountered issues very early on as there was no general way how to express edge-flow for training. We tried to reframe the issue into a reconstruction problem, walking the fine line of not only trying to achieve shape quality while also being aware of mesh quality.

This led us to point out and discuss the limitations of Chamfer distance, which was a limiting factor of other shape reconstruction research. We ended up resorting to predicting on point offsets, where we encountered another challenge in the form of acquiring ground truths for training. Computing shape offsets from primitive had shown to be a harder task than initially expected. We hinted at several approaches and discussed their boundaries while also reaching an algorithmic approach, which could solve the offset computation task. This approach was deeply involved in the field of graph theories, and we decided that its implementation was beyond the scope of this thesis. We made a workaround to show proof of concept for offset prediction by using Chamfer distance in a stochastic manner. Despite the limited ground truth quality, the computational results were good enough for us to continue.

We attempted to train a graph neural network to predict the offsets. Unfortunately, we were unsuccessful. Despite many attempts to improve the training, such as cleaning the dataset, changing the structure of the architecture, or tuning the optimizer's hyperparameters, nothing seemed to work. Each time, we were able to improve the training only slightly, but we were still unable to remove the obscure training defect that kept reappearing. Our model kept converging too soon or ended up in an overfitting state.

Regardless of the model's unsuccessful training, we fully implemented the proposed prediction pipeline along with the necessary data processing scripts. The code is easily executable, extendable, and documented if any investigation of why the training had failed were to occur.

We think the offset prediction can be utilized for edge-flow optimization if

we find a solution how to compute the offsets and fix the network's training. It would also be very beneficial if the solution did not expect the same number of connections and points between the input shape and the resulting shape. Meaning that the model would be able to add or remove a point and an edge. This way, we wouldn't have to deform a primitive but instead move to deform the input mesh while still preserving the shape. We would predict the new position of the points and either remove or add an edge between the points. Most importantly, we would still be operating on the initial shape, removing the shape reconstruction task from the prediction.

Bibliography

[1] Kozák, M. Static malware detection using recurrent neural networks. [cit. 2023-12-28]. Available from: https://dspace.cvut.cz/bitstream/handle/10467/88342/F8-BP-2020-Kozak-Matous-thesis.pdf?sequence=-1&isAllowed=y

[2] Science, T. D. A Comprehensive Guide to Convolutional Neural Networks - the ELI5 way [online]. Dec 2018, [cit. 2023-12-21]. Available from: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[3] Leskovec, J. Stanford CS224W: Machine Learning with Graphs [online]. Available from: http://web.stanford.edu/class/cs224w/,note={[cit.2023-9-20]}

[4] Sanchez-Lengeling, B.; Reif, E.; et al. A Gentle Introduction to Graph Neural Networks. *Distill*, 2021, doi:10.23915/distill.00033, https://distill.pub/2021/gnn-intro.

[5] Holzner, A. LSTM cells in PyTorch [online]. Oct 2017, [cit. 2023-12-28]. Available from: https://medium.com/@andre.holzner/lstm-cells-in-pytorch-fab924a78b1c

[6] Turk, G.; Levoy, M. Zippered polygon meshes from range images. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, New York, NY, USA: Association for Computing Machinery, 1994, ISBN 0897916670, p. 311–318, doi:10.1145/192161.192241. Available from: https://doi.org/10.1145/192161.192241

[7] Conway, B. What Is Mesh Decimation And Why Is It Vital For AR? [online]. [cit. 2023-10-1]. Available from: https://www.vntana.com/blog/what-is-mesh-decimation-and-why-is-it-vital-for-ar/

[8] Martin, J. Topology Guides [online]. [cit. 2023-10-15]. Available from: https://topologyguides.com/

[9] Reinhardt, Z. Retopology in Blender [online]. [cit. 2023-10-1]. Available from: https://www.youtube.com/watch?v=X2GNyEUvpD4

[10] Jakob, W.; Tarini, M.; et al. Instant Field-Aligned Meshes. *ACM Transactions on Graphics (Proceedings of SIGGRAPH ASIA)*, volume 34, no. 6, Nov. 2015, doi:10.1145/2816795.2818078.

[11] Jiang, B.; Zhang, J.; et al. BCNet: Learning Body and Cloth Shape from A Single Image. *CoRR*, volume abs/2004.00214, 2020, 2004.00214. Available from: https://arxiv.org/abs/2004.00214

[12] Jiang, C.; Huang, J.; et al. ShapeFlow: Learnable Deformation Flows Among 3D Shapes. In *Advances in Neural Information Processing Systems*, volume 33, edited by H. Larochelle; M. Ranzato; R. Hadsell; M. Balcan; H. Lin, Curran Associates, Inc., 2020, pp. 9745–9757. Available from: https://proceedings.neurips.cc/paper_files/paper/2020/file/6f1d0705c91c2145201df18a1a0c7345-Paper.pdf

[13] Wang, N.; Zhang, Y.; et al. Pixel2Mesh: Generating 3D Mesh Models from Single RGB Images. In *ECCV*, 2018.

[14] Gupta, K.; Chandraker, M. Neural Mesh Flow: 3D Manifold Mesh Generationvia Diffeomorphic Flows. *CoRR*, volume abs/2007.10973, 2020, 2007.10973. Available from: https://arxiv.org/abs/2007.10973

[15] Chang, A. X.; Funkhouser, T.; et al. ShapeNet: An Information-Rich 3D Model Repository. Technical report arXiv:1512.03012 [cs.GR], Stanford University — Princeton University — Toyota Technological Institute at Chicago, 2015.

[16] Toldo, R.; Castellani, U.; et al. A Bag of Words Approach for 3D Object Categorization. 05 2009, ISBN 978-3-642-01810-7, pp. 116–127, doi: 10.1007/978-3-642-01811-4_11.

[17] Chen, X.; Golovinskiy, A.; et al. A Benchmark for 3D Mesh Segmentation. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, volume 28, no. 3, Aug. 2009.

[18] Xu, J.; Cheng, W.; et al. InstantMesh: Efficient 3D Mesh Generation from a Single Image with Sparse-view Large Reconstruction Models. 2024, 2404.07191.

[19] McCulloch, W. S.; Pitts, W. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, volume 5, no. 4, 1943: pp. 115–133, ISSN 0007-4985.

[20] Chen, Y.-Y.; Lin, Y.-H.; et al. Design and Implementation of Cloud Analytics-Assisted Smart Power Meters Considering Advanced Artificial Intelligence as Edge Analytics in Demand-Side Management for Smart Homes. *Sensors*, 05 2019, doi:10.3390/s19092047.

[21] Bengio, Y.; Goodfellow, I.; et al. *Deep learning*, volume 1. Citeseer, 2017, ISBN 0262035618, 166–485 pp.

[22] Krishtopa. What Are Neural Networks, Why They Are So Popular And What Problems Can Solve [online]. 2016. Available from: `https://steemit.com/academia/\spacefactor\@m{}krishtopa/what-are-neural-networks-why-they-are-so-popular-and-what-problems-can-solve`

[23] Rosenblatt, F. The Perceptron: A Probabilistic Model For Information Storage And Organization In The Brain. *Psychological Re-view*, 1958: p. 2047, doi:0.1037/h0042519.

[24] Nielsen, M. A. *Neural Networks and Deep Learning*. Determination Press, 2015.

[25] Rojas, R. *Neural networks: a systematic introduction*. Springer Science & Business Media, 2013, ISBN 9783642610684, 37–99 pp.

[26] Leskovec, J.; Rajaraman, A.; et al. *Mining of massive data sets*. Cambridge university press, 2020, ISBN 9781108476348, 523–569 pp.

[27] Maladkar, A. I. M., Kishan. 6 Types of Artificial Neural Networks Currently Being Used in ML. [cit. 2023-12-25]. Available from: `https://analyticsindiamag.com/6-types-of-artificial-neural-networks-currently-being-used-in-todays-technology/`

[28] Lipton, Z. C.; Berkowitz, J.; et al. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*, 2015: pp. 5–25, ISSN 2331-8422. Available from: `https://arxiv.org/pdf/1506.00019.pdf`

[29] Wiki, B. M. . S. Feedforward Neural Networks [online]. [cit. 2023-12-25]. Available from: `https://brilliant.org/wiki/feedforward-neural-networks/`

[30] Towards AI, M. Main Types of Neural Networks and its Applications - Tutorial [online]. [cit. 2024-04-25]. Available from: `https://medium.com/towards-artificial-intelligence/main-types-of-neural-networks-and-its-applications-tutorial-734480d7ec8e`

[31] Goodfellow, I.; Bengio, Y.; et al. *Deep Learning*. MIT Press, 2016, url-http://www.deeplearningbook.org.

[32] Wiki, B. M. . S. Backpropagation [online]. [cit. 2023-12-21]. Available from: https://brilliant.org/wiki/backpropagation/

[33] Brownlee, J. How Do Convolutional Layers Work in Deep Learning Neural Networks? [online]. April 2020, [cit. 2023-12-25]. Available from: https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/

[34] MathWorks. Convolutional Neural Network [online]. [cit. 2023-12-21]. Available from: https://www.mathworks.com/solutions/deep-learning/convolutional-neural-network.html

[35] Science, T. D. Recurrent Neural Networks [online]. Jun 2019, [cit. 2023-12-21]. Available from: https://towardsdatascience.com/recurrent-neural-networks-d4642c9bc7ce

[36] IBM. What are Recurrent Neural Networks? [online]. [cit. 2023-12-25]. Available from: https://www.ibm.com/cloud/learn/recurrent-neural-networks

[37] Medium. Understanding Recurrent Neural Networks in 6 Minutes [online]. Sep 2019, [cit. 2023-12-21]. Available from: https://medium.com/x8-the-ai-community/understanding-recurrent-neural-networks-in-6-minutes-967ab51b94fe

[38] Olah, C. Understanding LSTM Networks [online]. [cit. 2024-2-28]. Available from: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[39] Hochreiter, S.; Schmidhuber, J. Long Short-term Memory. *Neural computation*, volume 9, 12 1997: pp. 1735–80, doi:10.1162/neco.1997.9.8.1735.

[40] Phi, M. Illustrated Guide to LSTM's and GRU's: A step by step explanation [online]. [cit. 2024-2-28]. Available from: https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

[41] Baumgart, B. G. Winged edge polyhedron representation. Technical report, Stanford, CA, USA, 1972.

[42] Predescu, A.-D.; Triantafyllidis, G. Real-time, anthropomorphic 3-D scanning and voxel display system using consumer depth cameras as an interactive means of individual artistic expression through light. *SHS Web of Conferences*, volume 43, 01 2018: p. 01002, doi:10.1051/shsconf/20184301002.

[43] Levoy, M.; Whitted, T. The Use of Points as a Display Primitive. 2000. Available from: https://api.semanticscholar.org/CorpusID:12672240

[44] Foundry. Instant Meshes algorithm - an interview with Dr. Wenzel Jakob [online]. [cit. 2023-10-15]. Available from: https://www.foundry.com/insights/vr-ar-mr/mitsuba-renderer-instant-meshes

[45] Chen, Z. A Review of Deep Learning-Powered Mesh Reconstruction Methods. 2023, 2303.02879.

[46] Wen, C.; Zhang, Y.; et al. Pixel2Mesh++: Multi-View 3D Mesh Generation via Deformation. In *ICCV*, 2019.

[47] Charles, R. Q.; Su, H.; et al. PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 77–85, doi: 10.1109/CVPR.2017.16.

[48] Nash, C.; Ganin, Y.; et al. PolyGen: An Autoregressive Generative Model of 3D Meshes. *CoRR*, volume abs/2002.10880, 2020, 2002.10880. Available from: https://arxiv.org/abs/2002.10880

[49] Knuth, D. E. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Reading, Mass.: Addison-Wesley, third edition, 1997, ISBN 0201896834 9780201896831.

[50] Bresson, X.; Laurent, T. Residual Gated Graph ConvNets. *CoRR*, volume abs/1711.07553, 2017, 1711.07553. Available from: http://arxiv.org/abs/1711.07553

[51] Brody, S.; Alon, U.; et al. How Attentive are Graph Attention Networks? *CoRR*, volume abs/2105.14491, 2021, 2105.14491. Available from: https://arxiv.org/abs/2105.14491

[52] Du, J.; Zhang, S.; et al. Topology adaptive graph convolutional networks. *CoRR*, volume abs/1710.10370, 2017, 1710.10370. Available from: http://arxiv.org/abs/1710.10370

Appendix A

Acronyms

ANN Artificial Neural Network

RNN Recurrent Neural Network

CNN Convolutional Neural Network

LSTM Long Short-Term Memory

ReLU Rectified Linear Unit

ResNetGate Residual Gated Network

Appendix B

Contents of enclosed CD

- `README.md` ..... the Markdown file with description
- `environment.txt` ..... list of necessary dependencies
- `source` ..... the directory of source files
  - `data` ..... the directory with training samples
  - `deformation` ..... the directory with deformation modul
    - `deform_mesh.py` ..... deformation related utilities
  - `pretrained_model` ..... the directory of pretrained model
    - `model.pth` ..... saved model structure
    - `model_weigths.pth` ..... pretrained weights
  - `training` ..... the directory of model training moduls
    - `dataset.py` .. dataset wrapping class for easy loading and iteration
    - `gnn_model.py` ..... definition of the model and its architecture
    - `trainer.py` ..... helper class to run training and evaluation
  - `utils` ..... the directory of utility moduls
    - `io_operations.py` ..... iIO related utility functions
    - `mesh_utils.py` ..... mesh related utility functions
  - `data_prep.py` ..... data preperation entry script
  - `predict.py` ..... prediction pipeline entry script
  - `test_target_bear.obj` ..... test bear sample for prediction
  - `test_target_cup.obj` ..... test cup sample for prediction
  - `training_model.py` ..... model training entry script