IGR Report: Neural Subdivision

by **Benjamin DUPUIS**

[Video presentation of the project]

I. Abstract

In recent years, subdivision surfaces have evolved from purely analytical, fixed-rule approaches into data-driven models that learn to adapt to local geometry. Based on the recommendation from my teacher Mr. Parrakat, I chose to work on ***Neural Subdivision*** (available here). This paper presents a subdivision framework where a neural network replaces the traditional linear averaging in Loop subdivision with a learned, non-linear rule. This innovation allows for the preservation of fine details and the introduction of stylistic variations—a quality that is especially attractive for creative applications.

Building on this work, my research has focused on further exploring the "stylized" subdivision aspect. In parallel, I have developed a Blender extension that integrates the neural subdivision framework directly into the popular 3D Software, enabling artists and technical designers to experiment with and adopt these advanced subdivision techniques more seamlessly. The goal of this report is to concisely present the work from the original paper in a clear and accessible manner, while also summarizing my own work and the challenges I encountered.

Unlike a traditional scientific paper, this report includes discussions of issues encountered during the development stages that may not directly impact the final results. Additionally, my work was exploratory rather than rigorously validated on large datasets, relying instead on a limited set of examples. I'll start by introducing and presenting the original paper, I'll proceed with the presentation of my work and the issues I faced. I'll finish with the literature review, that I chose to put at the end as it does not really serve the main content of this report and as, while being interesting, it is quite long and needs to build up on the presentation of the paper.

II. Paper presentation

First and foremost, I want to acknowledge the exceptional quality of this paper. The explanations are remarkably clear, well-motivated, and easy to follow. In this section, I will summarize the paper while omitting the finer details of the full implementation, as they are very well explained in the original paper. The paper ***Neural Subdivision*** introduces a novel, data-driven framework for coarse-to-fine mesh modeling that learns non-linear subdivision rules, rather than relying on the fixed linear averaging rules of classical methods, such as loop division seen in class. The goal is to overcome typical limitations of standard subdivision surfaces—such as volume shrinkage, over-smoothing, or amplification of tessellation artifacts—by infusing learned, geometry-aware, adjustments into the subdivision process.

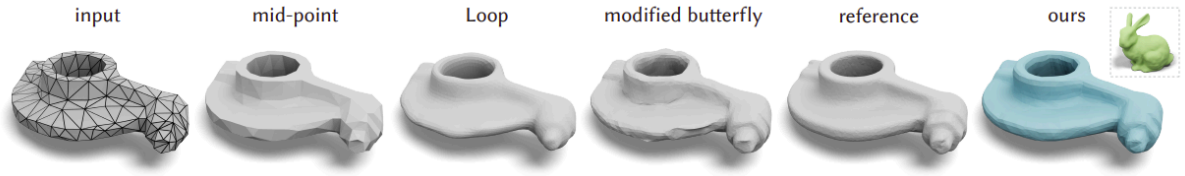

Fig 1:
Comparison of different subdivision methods on a single example.

a. Goal

The primary aim is to design a neural subdivision method that, during inference, accepts a coarse triangle mesh and recursively refines it via fixed topological updates (as in Loop subdivision) while using a neural network to predict vertex positions based on local geometric context. In doing so, the method is able to capture fine details and local variations that classical linear rules miss. Furthermore, the network is trained in a self-supervised manner using only high-resolution meshes. Training pairs are generated by decimating a high-res model using random edge collapses while simultaneously constructing a high-quality bijective mapping between the coarse and fine meshes. This mapping allows an efficient $\ell 2$ loss to be defined that respects the manifold structure of the surface.

b. Inner workings

At the heart of the approach is the idea of "neural subdivision": while the topology is updated using the standard mid-point splitting (as in Loop subdivision), the repositioning of vertices is handled by a neural network that learns a non-linear rule conditioned on local geometry. The network is architected around three sequential modules (schema below):

- Module I (Initialization): It computes high-dimensional per-vertex feature vectors by aggregating differential geometric quantities (for instance, the difference between a vertex's position and the average of its neighbors) from the 1-ring neighborhood. These features are represented in local frames to ensure invariance to rigid transformations.
- Module V (Vertex update): For each subdivision iteration, this module updates features at the original (or "even") vertices by aggregating information from surrounding half-flap patches. A half-flap is defined on a directed edge with its two adjacent triangles and provides a canonical ordering that simplifies feature extraction.
- Module E (Edge update): New vertices inserted at edge midpoints are repositioned by pooling differential features from the two half-flaps (i.e. both orientations of the edge) to produce their final position updates.

These modules are implemented as shallow multi-layer perceptrons (MLPs) with shared weights across all levels of subdivision. A key design decision is to operate entirely in a local, differential coordinate

space—this not only ensures that the process is invariant to rotations and translations but also means that even a network trained on a single exemplar (for example, the Stanford bunny) can generalize to novel meshes.

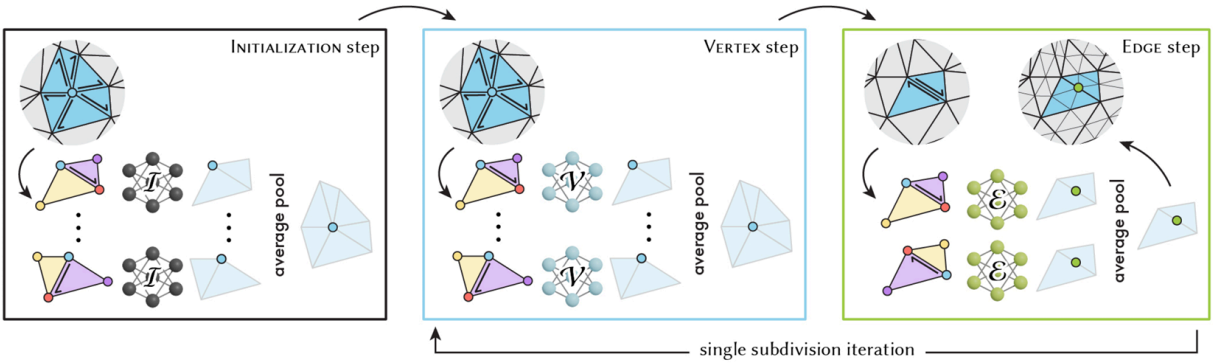

Fig. 18. Our neural subdivision consists of three sequential steps: Initialization , Vertex , and Edge , with three network modules: $\mathcal{I}$, $\mathcal{V}$, and $\mathcal{E}$ for each step respectively. In both Initialization and Vertex steps, we apply $\mathcal{V}$ and $\mathcal{E}$ for the half-flaps of all the outgoing edges of a vertex, and use average pooling to combine the output features back to the center vertex (blue). In the Edge step, we apply $\mathcal{E}$ to both half-flaps of an undirected edge and use average pooling to map the output features to the center vertex (green) of the edge.

On the training side, a novel data-generation scheme called "successive self-parameterization" is introduced. High-resolution meshes are decimated using random edge collapse sequences. During each collapse, the method computes a conformal parameterization of the local 1-ring (using fixed boundary conditions) so that a bijective mapping between the coarse and original meshes can be maintained. This bijectivity is critical for accurately assigning a target position to every new vertex during subdivision. The overall training loss is computed as an ℓ2 distance between the network's output and the corresponding ground truth vertex positions (obtained via the composite bijective map, I will not detail how the map is computed here), which has been shown to be orders of magnitude faster and more robust than chamfer distance based approaches.

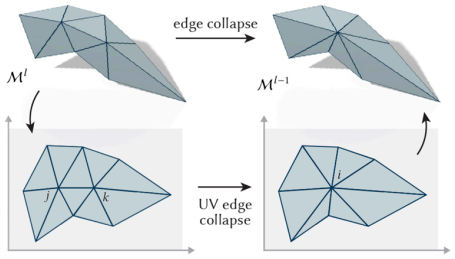

Fig. 13. For each edge collapse, we simultaneously collapse the edge on the 3D mesh (top) and the UV domain (bottom). As the boundary vertices of the edge's 1-ring are preserved through the edge collapse, we constrain the flattened boundary in UV space to be at the same position when computing the conformal parameterization of the post-collapse 1-ring.

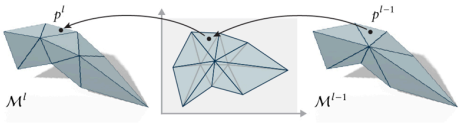

Fig. 14. Since both the pre-collapse and post-collapse parameterizations of the 1-ring map it into the same 2D domain, we can easily use the shared UV space to map a point back and forth between $\mathcal{M}^l$ and $\mathcal{M}^{l-1}$.

c. Results

Extensive qualitative and quantitative evaluations demonstrate the effectiveness of the proposed framework:

Category	$\mathbb{H}_{\text{loop}}$	$\mathbb{H}_{\text{m.b.}}$	$\mathbb{H}_{\text{ours}}$	$\mathbb{M}_{\text{loop}}$	$\mathbb{M}_{\text{m.b.}}$	$\mathbb{M}_{\text{ours}}$
Cat	2.75	2.17	**2.08**	0.73	0.21	**0.17**
David	2.95	2.13	**1.83**	0.88	0.27	**0.20**
Dog	3.26	2.32	**2.11**	0.84	0.31	**0.26**
Gorilla	4.53	3.17	**2.56**	1.27	0.48	**0.36**
Horse	5.87	4.53	**4.04**	1.51	0.50	**0.45**
Michael	3.88	2.71	**2.24**	1.12	0.38	**0.28**
Victoria	4.25	3.01	**2.36**	1.12	0.39	**0.30**
Wolf	2.83	1.74	**1.63**	0.69	0.23	**0.21**

Category	$\mathbb{H}_{\text{loop}}$	$\mathbb{H}_{\text{m.b.}}$	$\mathbb{H}_{\text{ours}}$	$\mathbb{M}_{\text{loop}}$	$\mathbb{M}_{\text{m.b.}}$	$\mathbb{M}_{\text{ours}}$
Cat	2.75	2.17	**2.09**	0.73	0.21	**0.16**
Dog	3.26	2.32	**2.12**	0.84	0.31	**0.25**
Gorilla	4.53	3.17	**2.89**	1.27	0.48	**0.34**
Michael	3.88	2.71	**2.15**	1.12	0.38	**0.27**
Victoria	4.25	3.01	**2.49**	1.12	0.39	**0.28**
Wolf	2.83	1.74	**1.65**	0.69	0.23	**0.20**

– The neural subdivision method consistently produces refined meshes that more closely approximate the ground truth high-resolution surfaces compared to classical approaches like Loop or modified butterfly subdivision. Quantitative metrics—such as reduced Hausdorff distance (H) and lower mean surface distance (M) on benchmarks (e.g. using the TOSCA dataset)—confirm these improvements.

Fig 3: Training on a single category, Centaur (top table), and three categories, Centaur, David, Horse (bottom table), separately, and evaluate by subdividing the rest of the TOSCA shapes.

– The approach shows strong generalization capabilities. Even when trained on a single exemplar, the network is able to produce high-quality subdivisions on unseen shapes, including those with significant non-isometric deformations. It even adapts well when the input coarse mesh is generated by alternative decimation techniques or artistic modeling, preserving manifold structure and local geometric details.

– An interesting by-product is the ability to perform “stylized” subdivisions. Training on different types of shapes (e.g., smooth organic versus man-made objects with sharp features) leads the network to generate subdivision results that are biased toward the style of the training data, a tool that could be used for design and artistic applications.

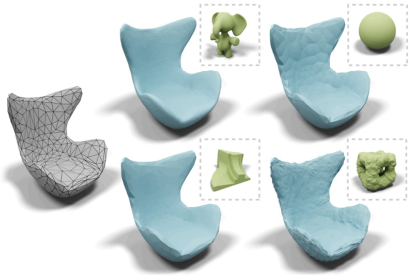

Fig 4: Using different shapes in training leads to stylized subdivision results (blue) biased towards the training shapes (green)

– A key limitation of this approach is its exclusive reliance on local geometric features, which, while beneficial for ensuring invariance to rigid transformations, restricts its ability to capture global semantic context and coherent overall structure. Additionally, the network does not incorporate an explicit scaling factor, meaning that although it excels at refining local details, it struggles (a lot) to generalize across objects of significantly different sizes. Lastly, it is only meant for triangle meshes, and a pre-processing step must be applied for objects using other geometries.

III. My work

This section details my personal exploration and development of the Neural Subdivision framework. This is not a report in the traditional sense, more of a chronicle of my efforts and my results.

a. Motivation

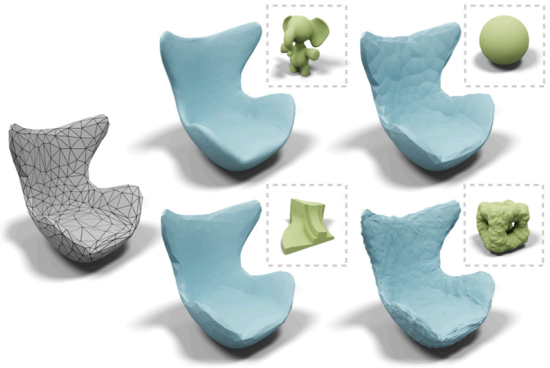

Fig 4: Using different shapes in training leads to stylized subdivision results (blue) biased towards the training shapes (green)

In the initial paper, the impact of the initial shape on the result seemed really interesting to me. The paper does not elaborate much on it, but it seemed to me that, if used properly, this fact holds potential to create a practical (or at least fun and enjoyable) tool to subdivide shapes according to a specific style. To be more precise, I had two ideas in mind:

- It might be possible to speed up the process of creating a series of similar detailed meshes by meticulously detailing a few, training the network on them, and then applying the learned model to the coarse versions of the remaining shapes.
- It might also be possible to "stylize" a mesh by using the network, possibly by first decimating it and then re-subdividing it using a network trained to achieve a desired style effect.

b. Code provided by the paper author

The authors of the paper decided to make their code (python and MATlAB) public on this github. They included basic instructions on how to train and test the network. This is where I began. I cloned their repository, and ran the provided example network on a few meshes. Right away, I noticed a major flaw mentioned in the paper: the network performs poorly when the triangle sizes deviate significantly from those in the training data. This issue becomes especially apparent when running subdivision on an already subdivided mesh, leading to numerous artifacts.

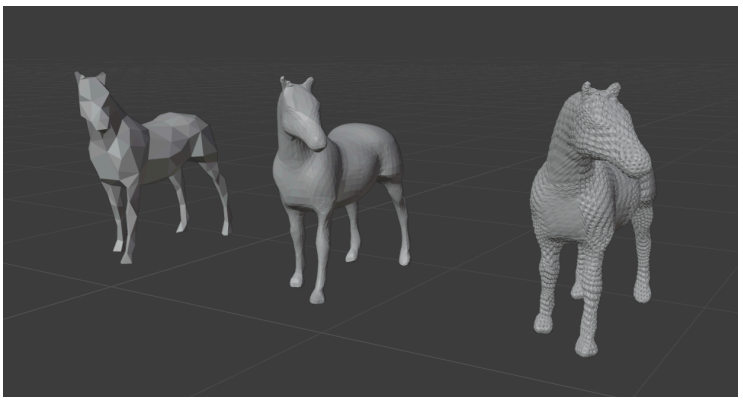

Fig 5: original mesh (left), after one round of subdivision (middle) after two rounds (right)

Another challenge I encountered—and which I will discuss in more detail later—was the network's slow performance on meshes with a large number of faces. It handles thousands or even tens of thousands of faces reasonably well, but anything beyond that can take minutes to process.

I then attempted to train a network using two small .obj files (each containing approximately 15,000 triangles). The training was completed successfully in about 30 minutes. However, the mesh preprocessing stage took a significant amount of time. The paper's authors acknowledged this limitation, noting that their MATLAB code (used for generating decimated meshes) was quite slow. To address this, they linked to another repository containing a faster C++ implementation, which I will explore later.

At this point, while I felt I had a solid understanding of the model's underlying architecture, I also found the repository structure and the number of required steps for training a network to be unnecessarily complex. I plan to simplify these later.

c. First project idea: c++ reimplementation

Rather than just experimenting with the provided network, I wanted to take a more code-heavy approach that better aligned with our coursework. My first plan was to reimplement the network in C++. Having previously worked with LibTorch (PyTorch's official C++ API), I felt quite confident and knew this approach would offer several advantages:

- **Faster mesh processing** – Working with mesh is way faster with C++. Python libraries such as trimesh are really slow in comparison.
- **Potential speed improvements** – while LibTorch and PyTorch take roughly the same time for inference and gradient computation, C++ runs noticeably faster for the rest of the code.
- **Seamless integration** – incorporating the faster C++ implementation for mesh preprocessing mentioned earlier.
- **Real-time visualization** – This is the main point. I'd be able to easily use OpenGL to visualize the network's results directly. I could also keep the current code, but I want to limit the use of endpoints between different languages as much as possible.

However, and for reasons that to this day are still unclear to me, my computer refused to compile the project when I had both OpenGL and LibTorch imported. I think it has something to do with the fact that I use LibTorch for ROCm (AMD's CUDA equivalent, which is extremely bad and unsupported, I did not know I liked deep learning that much when I bought my GPU). After a week-end spent trying to fix this issue, a Linux full reinstall to try it on Ubuntu and the discovery that my Fedora (dual-boot with Windows) only had

access to two cores of my CPU (out of 6?) I decided that I spent too much time on something that mainly interested me because it was C++ code, so I could prove to myself that I improved and was capable of writing fully working programs on my own.

d. Second idea: Integrating it as a blender extension

My goal now was to start experimenting with the stylization and different training of the networks. But to at least write a bit of code and make it more friendly for me to use, I decided to integrate the network into blender. As such, I set my mind into developing a Blender Extension that would allow me to use the network from within the application. I also thought, and still think, that learning how to make a proper extension for an application (I only developed scripts before) was a valuable skill that could only be great to learn. I started by trying to follow blender's official documentation, but it is very scarce and does not provide a proper beginner tutorial on how to start building an extension. With the help of online tutorials, I quickly managed to get a working extension that does simple things such as scaling up and down an object, retrieving any object as an obj file, working with vertices, …

I initially thought I only needed to adapt my existing Python files, but it turned out to be more complex. Installing the required libraries (PyTorch, SciPy, and NumPy) was not as simple as running pip install torch scipy numpy in Blender's console. After following multiple tutorials, I started using Poetry, which I had seen before but never learned. It is said to be very useful for managing dependencies, ensuring compatibility, versioning and packaging. I also had to learn what python wheels are and how to build them (I often get wheel errors when using Python but never properly learned what they were). All this process took a lot of time (in the tens of hours), but once I managed to set it up properly, I quickly reimplemented the original code inside the extension. I also had to make sure the meshes are triangulated before they can be fed to the network. At this point it is possible, from within the app, to choose a network among the ones packaged in the extension and run it on any mesh in a blender project.

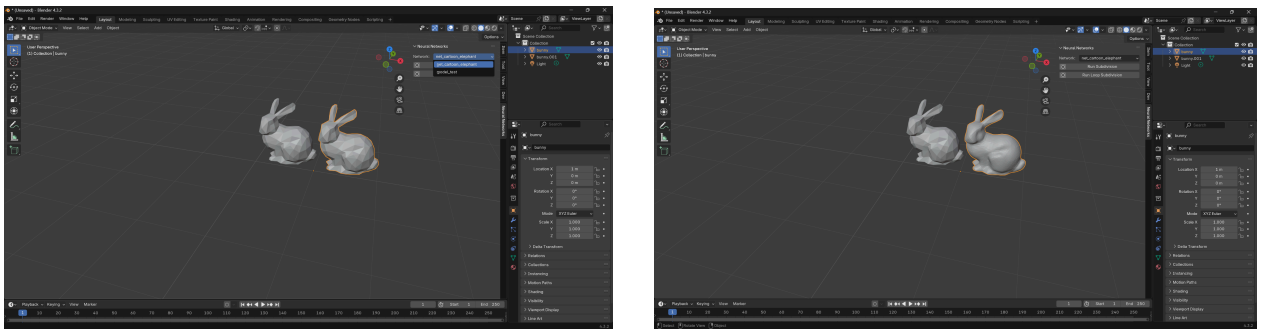

Fig 6: Interface to select the network (left) and the ran algorithm, using the button or the keybind SHIFT + N (right)

Now is the right time to mention that there are still a few caveats: First, my extension does not permit training the network. Also, because it had to include all the wheels, the extension .zip file takes up 2.4GB. Furthermore, I only built the wheels for windows, as it is now the system I'm using for this project. A proper extension should build a version for all the main OS versions. Lastly, because I'm on Windows and PyTorch is not using my GPU (no PyTorch support for AMD GPUs on Windows), the application of the network is quite slow. While almost instant for the rabbit (400 edges), it takes a minute for bigger objects (5000 edges), and it seems to be almost linear from there (10 minutes for 50000 edges).

For the training process, I created a modified version of the original repo. I created a python script that takes a file containing a few hyperparameters as an input and returns a network. To preprocess the data, I compiled a slightly modified version of the repo replacing the MATLAB script and ran it from my python file. Basically, you only need to enter the path to a folder containing the .obj files you want for your

training, and the script does all the processing and returns the network in a format compatible with the extension.

e. Experimentation to bias the network towards better reconstruction

The first thing I tried is to bias the network with shapes similar to the one we want to obtain in order to improve the results. Here is an example: On the left, I have my training shapes. I tried to only pick very humanoid busts of different qualities. On the right the results I got on an example. The original mesh is on the left, the one obtained with subdivision in the middle and the one obtained with my network on the right. Overall, the result is quite good. I would argue the result obtained by my network is better: For a resulting similar number of faces, the hairs look better on the right side to me, but the triangles are “bigger” on the face. However, we can still see a few artefacts on the side and at the bottom. The thing I still don’t quite understand is why on this specific example (it is not the case on others), the artifacts of the network subdivision look so similar to those of the loop subdivision.

Fig 8: Dataset used for training (left) and results obtained -left to right: original, loop and network- (right)

I would like to present more interesting results, but the time required for training models and the time I lost on the previous steps did not allow me to train many networks with complex objects. As such, I only had this one that was pertinent enough to be featured in the final report.

f. Experimentation to bias the network towards stylistic features

The second experiment I wanted to make is to train on objects presenting distinct traits and see how it would affect the output. To be more precise, I wanted to see if I could infer some kind of “style” from the training data and use this “style” on other meshes that do not have it. I mentioned above a pipeline that would consist of starting from a detailed mesh, degrade it, and rebuild it using this new style. In the end I did not implement that in the extension. I start from already decimated meshes (using blender’s decimate feature or found on the internet). Here are a few examples of datasets that I used for the training:

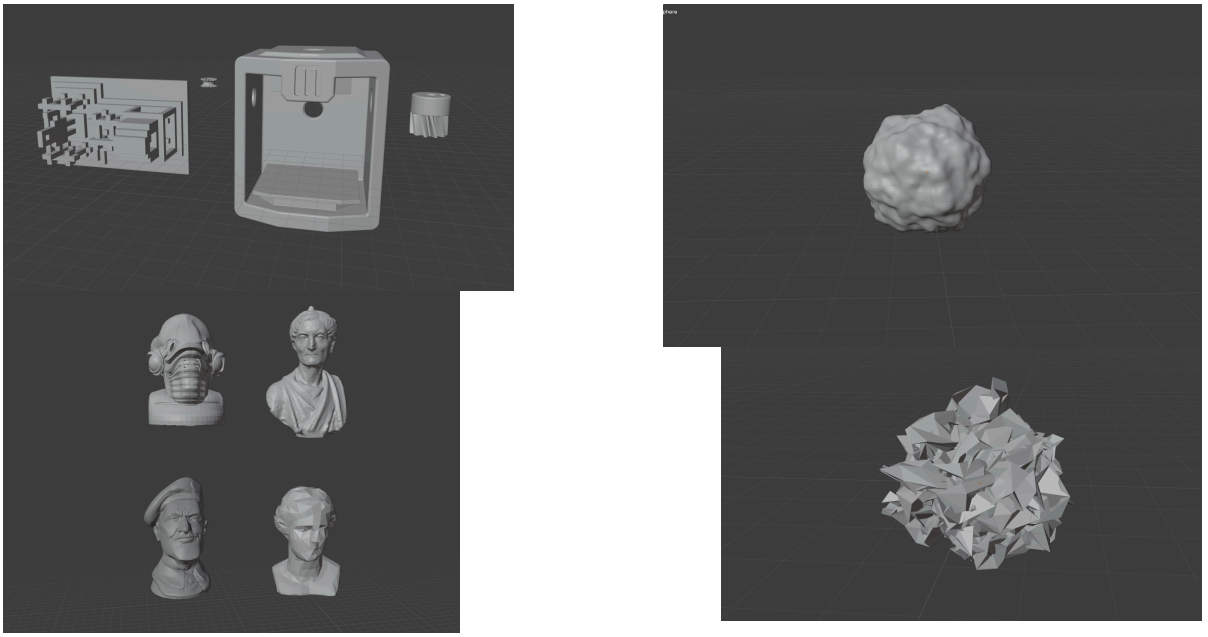

Fig 9: A few datasets I tried. A very mechanical one (top left), a simple organic sphere (top right), the same one as before (bottom left), a sphere that I tried to make as chaotic as possible using fractal noise (bottom right)

The quality of the results I obtained vary and are honestly a bit disappointing. In some cases, it is horrendous (horse below, even if it makes sense in this case).

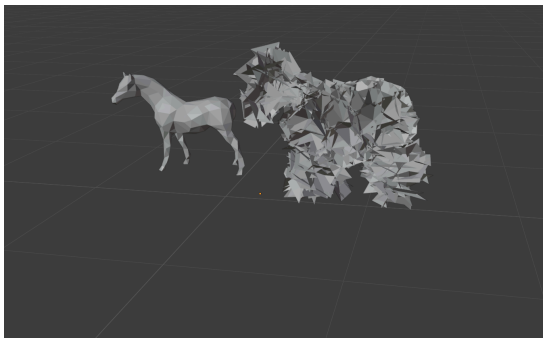

Fig 10: Subdivision of a simple horse mesh using the fractal noise as training

With only the few examples that I have I can see some results (rabbit below). I especially like the sphere one and the organic one. I can imagine situations when the two of them can be interesting.

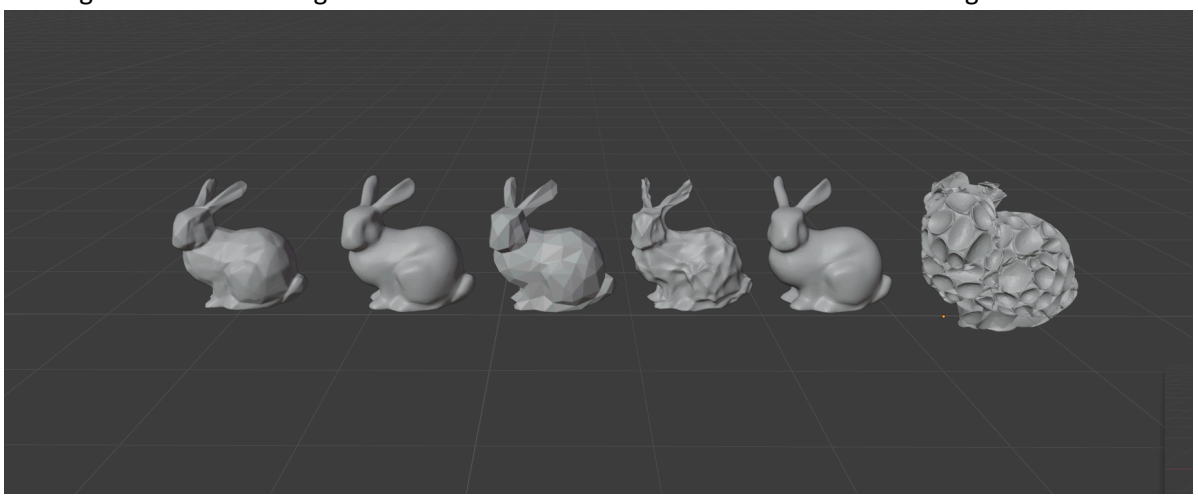

Fig 11 (left to right): Normal sphere + loop, humanoid + loop, original, organic sphere, loop, noise sphere + loop

Once again, I do not want to flood the report with examples for the very few models I trained.

g. Further improvements possible

My primary goal would be to train the model on additional datasets and test it on a wider range of examples. The results are not perfect, but they are satisfactory and I think a bit more work would have allowed me to get even better results. This is true for the subdivision based on similar shapes, where the training is very time-consuming, but also for stylized subdivision, where I could have found more ideas for interesting styles. Currently, my extension also has several limitations. The build process only compiles the extension for my operating system, but I could modify my script to automatically build it for Linux and Mac as well. Additionally, the training process runs separately from the extension. While integration is possible, my initial attempt caused Blender to freeze until the process completed, and I did not really try to solve this issue.

IV. Conclusion

This project explored the implementation and application of Neural Subdivision, a data-driven approach to mesh refinement. While the initial goal was to delve deeply into C++ reimplementation, challenges encountered shifted the focus towards integrating the existing Python codebase into a Blender extension. This pivot allowed for a more practical exploration of the method's potential, particularly in stylized subdivision. I would have liked to delve deeper in these topics, but the time spent on making the environment and the code work did not allow me to do so. The resulting Blender extension, though still in an early stage, provides a user-friendly interface for applying neural subdivision to 3D models.

While my direct contribution to C++ code was limited, and therefore the project might not fully align with the traditional expectations of the course, I feel I significantly improved in dependency management, cross-platform compatibility, and Python extension development. My understanding of neural network applications (which interest me a lot) has also been widened and my Blender skills have improved. Additionally, exploring a broader range of training datasets and refining the network's ability to capture stylistic features remains very interesting and I’m sure even better automatic tools for stylization will be created in the next few years. Lastly, the literature review (next page) highlights the ongoing evolution of neural subdivision techniques, and it might be the thing I enjoyed the most doing as it allowed me to explore the topic of mesh processing and generation using deep learning (a field that I never properly explored before).

V. Literature review

In this review, I use "our" for clarity reasons (using "this paper" makes it hard to read). Note that the first three paragraphs are basically a summary of the analysis of the background work proposed in the paper. The following paragraphs talk about more recent developments that happened after / in parallel to this paper.

Our approach builds on **classic subdivision surfaces** and advances in **neural geometry learning**. Subdivision techniques refine meshes recursively, with classic methods like Catmull-Clark (Catmull & Clark, 1998) [1] and Loop subdivision (Loop, 1987) [2] relying on linear averaging of neighboring vertex positions. These methods are widely used but tend to smooth out details. Theoretical research has explored their continuity and limit surfaces (Stam, 1998; Zorin, 2007) [3][4]

Non-linear subdivision methods have been studied to maintain geometric properties like smoothness and planarity (Bobenko et al., 2020) [5]. Recent work (Preiner et al. 2019) [6] introduced probabilistic shape refinement, but our approach instead trains a neural network to learn a data-driven subdivision scheme that generalizes across different geometries.

In **neural geometry** learning, generative networks have improved shape reconstruction (Tatarchenko et al., 2019) [7] and point cloud upsampling (Li et al., 2019) [8]. However, point-based methods lack connectivity information, requiring expensive post-processing (Kazhdan & Hoppe, 2013) [9]. Our approach avoids this by directly operating on meshes. Compared to deformation-based mesh generation (Groueix et al., 2018) [10], our method refines surfaces while preserving topology, making it more flexible for diverse inputs. By combining neural learning with classical subdivision rules, our approach generates high-quality subdivisions that outperform traditional methods in both fidelity and adaptability.

At the same time, neural network-based approaches have significantly expanded the scope of subdivision techniques. **Graph neural network (GNN)–**driven methods, as presented in a recent paper (Chen et al., 2024) [11], performs much better than Neural Subdivision (our paper) by capturing both local geometric details and global connectivity from learning using local triangular mesh patches (instead of the half-flaps here). On a connex (but not quite the same) topic, (Chen et al., 2023)[12] use a subdivision-based encoder–decoder architecture to be able to transmit and reconstruct meshes efficiently. This method proved to work better than training a model on said shapes using our method, and then using it to reconstruct the shapes.

Advances in **mesh neural networks** have also shown promise in handling irregular or non-manifold meshes. For example, methods based on dual graph pyramids (Li et al., 2024)[13] create hierarchical representations that allow robust feature extraction and aggregation over complex mesh structures. This approach facilitates adaptive subdivision even in the presence of connectivity defects or intricate geometries, extending the applicability of subdivision methods to a wider range of real-world data. Moreover, data-driven bi-directional lattice customization techniques (Liu et al., 2024)[14] have successfully integrated subdivision modeling into material design and optimization, demonstrating the versatility of these methods beyond conventional shape reconstruction (in this instance, only Catmull-Clark is used, by we can imagine most advanced techniques will be used in the future).

In summary, recent literature on mesh subdivision highlights a clear trend toward integrating classical geometric techniques with modern neural and data-driven methods. Our approach, which unifies these concepts, contributed to this evolving landscape by offering a framework that is both structurally aware and geometrically precise.

Citations:

1. Catmull, E., & Clark, J. (1998). Recursively generated B-spline surfaces on arbitrary topological meshes. *ACM Transactions on Graphics*, *18*(3), 183–188.
2. Loop, C. (1987). Smooth subdivision surfaces based on triangles. *Master's thesis, University of Utah, Department of Mathematics*.
3. Stam, J. (1998). Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values. *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, 395–404.
4. Zorin, D. (2007). A method for analysis of C^1-continuity of subdivision surfaces. *SIAM Journal on Numerical Analysis*, *45*(4), 1722–1736.
5. Bobenko, A. I., Pottmann, H., & Rörig, T. (2020). Multi-Nets: Classification of discrete and smooth surfaces with characteristic properties on arbitrary parameter rectangles. *Discrete & Computational Geometry*, *63*(3), 624–655.
6. Preiner, R., Steinberger, M., Wimmer, M., & Leimer, F. (2019). Continuous non-linear subdivision surfaces. *ACM Transactions on Graphics*, *38*(6), 1–13.
7. Tatarchenko, M., Dosovitskiy, A., & Brox, T. (2019). Single-view to multi-view synthesis. *International Journal of Computer Vision*, *127*(12), 1636–1650.
8. Li, R., Li, X., Fu, C.-W., Cohen-Or, D., & Heng, P.-A. (2019). PU-GAN: A point cloud upsampling adversarial network. *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 7202–7211.
9. Kazhdan, M. M., & Hoppe, H. (2013). Screened Poisson surface reconstruction. *ACM Transactions on Graphics*, *32*(3), 29:1–29:13.
10. Groueix, T., Fisher, M., Kim, V. G., Russell, B. C., & Aubry, M. (2018). 3D-CODED: 3D correspondences by deep deformation. *Proceedings of the European Conference on Computer Vision (ECCV)*, 235–251.
11. Chen, G., & Wang, R. (2024). Triangular Mesh Surface Subdivision Based on Graph Neural Network. Applied Sciences, 14(23), 11378. https://doi.org/10.3390/app142311378
12. Chen, Y.-C., Kim, V. G., Aigerman, N., & Jacobson, A. (2023). Neural progressive meshes. SIGGRAPH 2023. arXiv. https://doi.org/10.48550/arXiv.2308.05741
13. Li, X.-L., Liu, Z.-N., Chen, T., Mu, T.-J., Martin, R. R., & Hu, S.-M. (2024). Mesh neural networks based on dual graph pyramids. *IEEE Transactions on Visualization and Computer Graphics, 30*(7), 4211–4224. https://doi.org/10.1109/TVCG.2023.3257035
14. Liu, F., Wu, H., Wu, X., Xiang, Z., Huang, S., & Chen, M. (2024). Data-driven bi-directional lattice property customization and optimization. Materials, 17(22), 5599. https://doi.org/10.3390/ma17225599