3D Reconstruction of Point Clouds via Neural SDFs

Gene Chou
gchou@princeton.edu

Herman Ishengoma
hermani@princeton.edu

Deus Alleluia Nsenga
dnsenga@princeton.edu

Abstract

We explore representation capabilities of neural Signed Distance Functions (SDF) through an accessible and interactive interface. Specifically, we investigate lightweight models such that reconstructions can be visualized in seconds on a cpu in a modern laptop. We build three models with varying complexity, and analyze tradeoffs among speed, reconstruction quality, and generalization. In an attempt to provide a more interactive experience, we allow users to upload and process meshes through filters, as well as select number of points and resolutions for determining outputs. Overall, our experiments demonstrate the benefits and disadvantages of each of the three models, and we discuss their potential applications given these properties.

1 Introduction

Learning 3D representations of scenes is fundamental for computer vision [1], robotic manipulation [2], scene understanding [3], and content generation [4]. Implicit neural representations which employ neural networks to approximate 3D geometry have become popular. Existing methods have achieved lower memory requirements [5, 6] and faster training and inference speeds [7, 6] than conventional explicit representation methods [8, 9]. For instance, voxels without acceleration structures suffer from cubical memory requirements [8, 10, 11]. Point cloud approaches lack adjacency information [9] and optimizing mesh representations is difficult [12].

Neural Signed Distance Functions (SDF) [13] implicitly represent a surface with a function that evaluates to 0 when given a point on the surface, and otherwise return the signed distance to the surface. Existing methods are capable of approximating diverse synthetic geometry, for multiple objects [13, 14, 15] as well as for varied levels of detail for single objects [5, 6].

In this project, our main goal is to explore the representation and real-time inference capabilities of SDFs through an interactive interface. Specifically, we build three different models of varying complexity. The first and most lightweight model is a simple 8 layer fully connected network [13, 7, 14] that is trained to overfit on one shape. The second model follows [14]. We use the same 8 layer architecture, but during training we compute second-order gradients in a meta-learning fashion. This forces the model to generalize to different shapes through fast adaptation even without shape priors. During inference, the model adapts to an input point cloud in 5 iterations, and remains relatively fast. Our third model is an auto-encoder. Our decoder is the same 8 layer network, but we add a PointNet [9, 16] to learn shape priors. We perform local pooling for extracting local features and add a UNet for spatial localization [16, 17]. This leads to favorable generalization even on unseen classes, but requires significantly more time during testing. We provide a more detailed explanation of our architecture and training methods in Section 4, and some visual and timing results in Section 5.

To achieve smooth user interaction, we build our frontend in JavaScript similar to previous assignments. We let users select meshes and different filters for interaction. Then, by selecting the number of points and resolution, they can view reconstruction results and explore the tradeoff among speed, number of points for input, and model complexity. Furthermore, we deploy using Heroku, a cloud application platform, such that users do not need to worry about installing packages and low-level details.

Our main takeaways for this project include:

- We build a platform that allows users to explore the tradeoff among speed, number of points for input, and model complexity of Signed Distance Functions (SDF) without having to install packages or deal with low-level details.
- Our interface provides options to upload meshes, process them using filters, and view their reconstruction results for a complete user experience.
- Each of our three models have benefits and disadvantages that make them suitable for different applications.

2 Methods: Frontend – Creating an Interactive Interface

Our project is divided into two distinct parts: the Front End, written in Javascript, and the Back End, written in Python. These two parts of the project are connected to each other via Flask, executed by Heroku. The primary purpose of the Front End is to enable the user to view the meshes, edit the mesh via 'Filters', and to be able to randomly generate 'Point Clouds.' The Point Clouds, along with its corresponding data, made by the user are then sent to the Back End as a json file. The Backend then outputs a reconstructed mesh. Once completed the mesh is sent back to the Front End and displayed to the user.

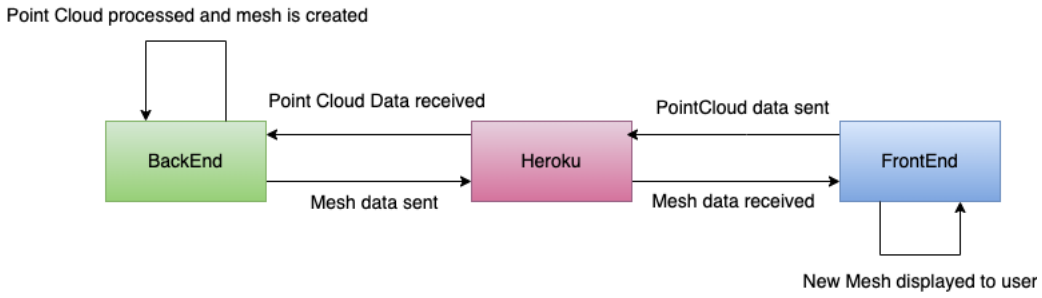
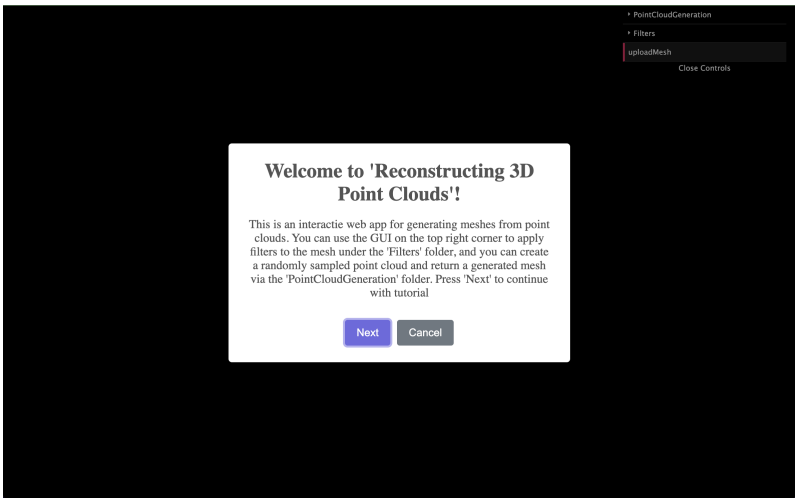
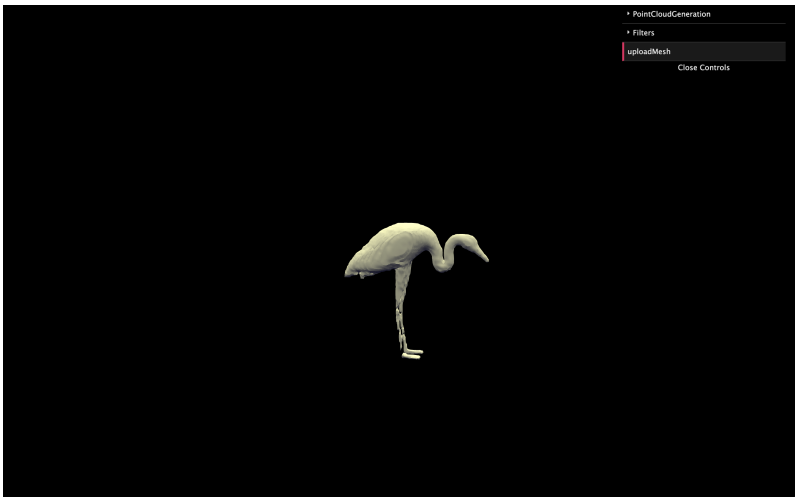


Figure 1: Diagram of overall architecture of the project

When one first accesses the web app, they are welcomed to a pop-up that serves as the Guide for the web app. Pressing 'Next' on the pop-up enables the user to continue with the Guide, whilst pressing 'Cancel' enables them to just start using the web app directly. Once the user exits or completes the guide, they are presented with a mesh that we already have stored for the user. The user is then able to replace this mesh with their own mesh if they so wish and edit the mesh in the scene with the filters and generate the Point Clouds and see the resulting mesh generated.



(a) Web App when user first accesses it



(b) Image of when the Guide/Tutorial has been closed/completed

Figure 2: An example of how the web app looks when one first accesses it

We opted to use the starter code as the foundation of our project as the starter code already enabled the user to observe the rendered mesh from different angles which is one of our primary goals. The starter code as well gave us a direct path to adding features we desire the user to have. An overview of how the main classes of the Front End interact with each other is shown in the diagram below.

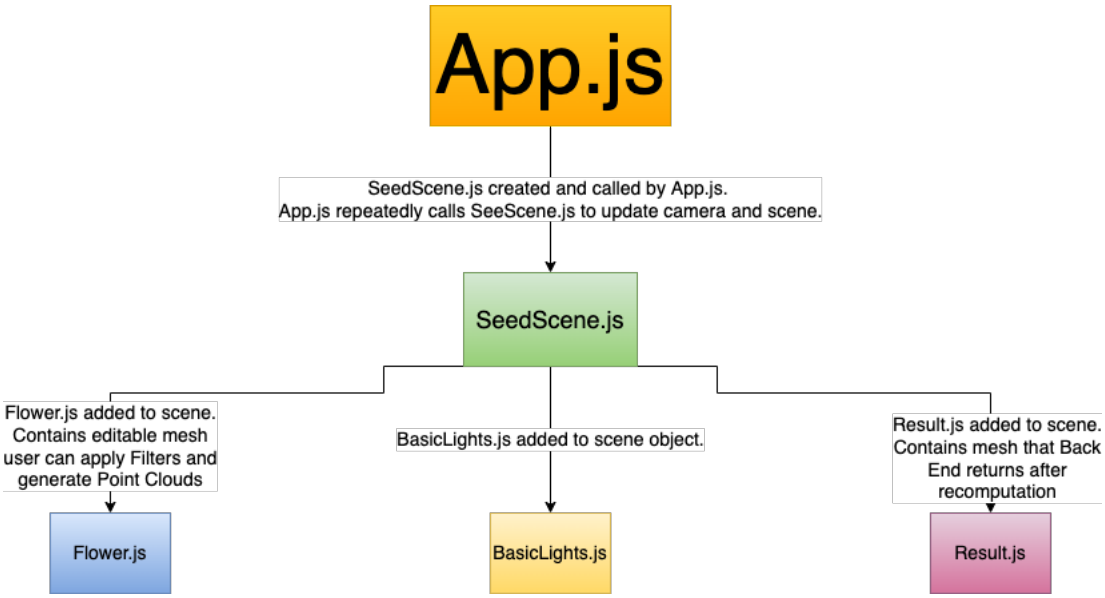


Figure 3: Diagram of architecture of Front End

App.js renders all the objects in the view using the Three.js library. SeedScene.js uses the Three.js library to create the scene and uses the dat.gui library to create the GUI. The SeedScene class contains all the objects that are in our scene including the lights, mesh that we generate the point clouds from, and the resulting mesh that we return from the Back End. Flower.js and Result.js contains the editable mesh and the resulting mesh that we return from the Back End, respectively.

The primary features/functions we implemented on the Front End are as follows:

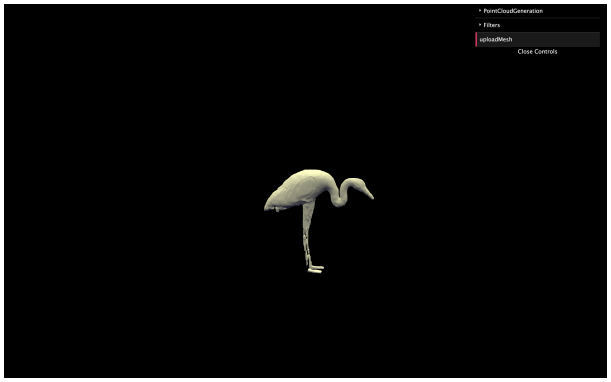
1. Rendering and uploading of Mesh(es)
2. Filters
3. Random Point Cloud Generation

2.1 Rendering and uploading of Mesh(es)

Of the meshes that we have readily stored for the user, all are of the .ply file type. As well, the file types that the user can upload are constrained only to the .ply file types. Whilst, Three.js recommends using .gltf file extensions [18], we opted to do this because the Back End worked with the .ply file types before we had the Front End. This was decided since the .ply file type is a standard file extension for meshes.

All meshes are loaded into the project using the PLYLoader loader from Three.js. From here, we then ensure that all the vertices exist within a bounding box with min values [-1,-1,-1] and max values [1,1,1]. This is because the Back End code works with meshes of this size. Hence, we scale down the mesh if it is too big.–Note: the scaling is only done on the mesh the user can edit, the mesh we retrieve from the Backend is not scaled so as to display the most truthful results of the inference.– We then opt to create a material of the class MeshStandardMaterial from the Three.js library. Initially, we had used a material of the MeshBasicMaterial class from the Three.js library, however, this class did not allow the user to see the finer details of the mesh. A demonstration of this is shown in the figure below. Once all this has been computed, we then render the mesh.

(a) Image of a mesh rendered having MeshBasicMaterial



(b) Image of a mesh rendered having MeshStandardMaterial

Figure 4: It is evident on the finer details that one can observe on image (b), wih the MeshStandardMaterial vs image (a), the MeshBasicMaterial

Finally, we opted to just use a single hemisphere light source into the scene using the Three.js library. We found this to be the best lighting setup that best illuminates the meshes and enables the user to observe the differences between the mesh they sampled the Point Clouds from and the mesh that was outputted by the Back End.

2.2 Filters

The project has five filters that one can apply to edit the mesh. The filters being: 'Noise', 'Inflate', 'Twist', 'Smooth', and 'Sharpening.' We opted for these features over others as given time constraints, we found it too difficult to implement features that would change the number of vertices and/or faces. So we implemented these features which did not change the face and vertex numbers.

Each feature can be applied via the GUI. For each feature we have a Slider(s) that enables the user to change the values of variables that a given Filter takes in. Once the user has the slider set to the desired variables, they can select the corresponding apply button for the given filter to apply the filters. For each filter we have applied we check if the applied filter changes the dimensions of the mesh such that its vertices are outside the bounding box of min [-1,-1,-1] and max [1,1,1]. If it does, the filtering operation applied, remains on the mesh, but the mesh is scaled so that it is still within the bounding box.

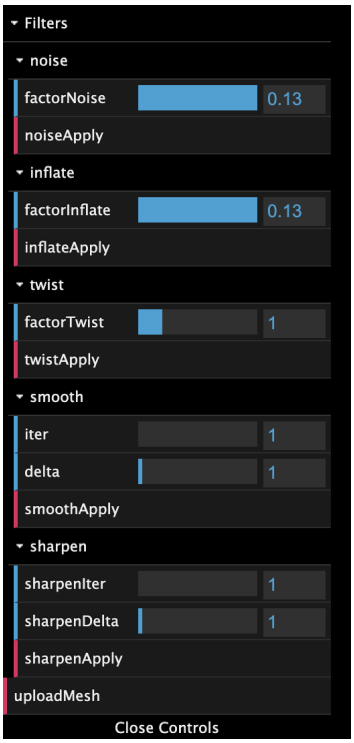
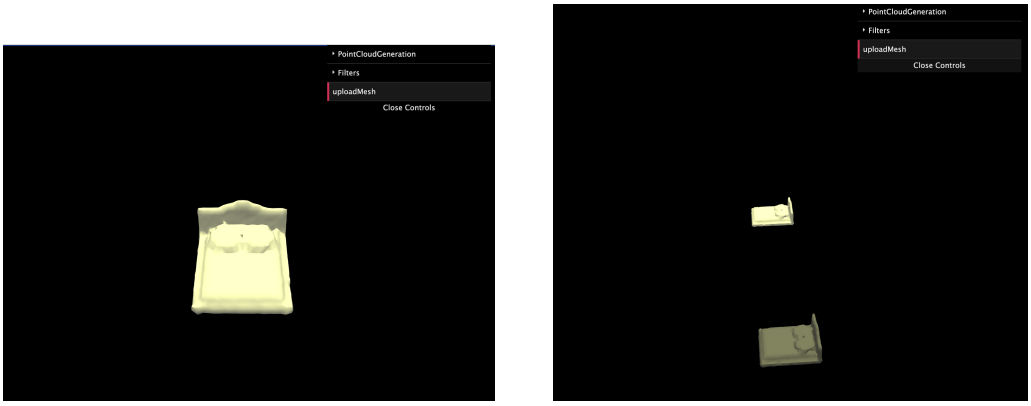


Figure 5: Image of the Filters that one can update. For instance in sharpen, the user can change the number of iterations, and delta values to what they wish and then select SharpenApply when they want to apply the sharpening filter.

2.3 Random Point Cloud Generation

As an overview, this feature would obtain from the user the size of the Point Cloud that the user wants to generate, the resolution the user wants to display the reconstructed mesh at, and the model type the user desires (details of the meaning of the model type is explained in **Section 2.3.3** and further in **Section 4.1**). Once these fields are provided by the user, the Frontend code samples 30,000 vertices from the mesh, if the mesh has less than 30,000 vertices we resample vertices from the mesh. These 30,000 vertices serve as ground truths for the mesh which allow us to provide to the user with the Chamfer Distance which is a quantitative metric that provides a score of how accurate the reconstructed mesh is from the original. Once the 30,000 vertices have been sampled, the Frontend, sends to the Backend 3 objects: the 30,000 vertices, the size of the Point Cloud the user wants to generate and the mesh type the user opted for. – The Point Cloud is generated by the Backend from the 30,000 vertices that the Frontend sent.– The Backend then processes the information and returns a reconstructed mesh that the Frontend then display.



(a) Initial interface **(b)** Showing reconstruction

Figure 6: When we initialize the interface, we load a mesh by default (a). Then we sample point clouds from the mesh and display its reconstruction (b). Users can freely zoom in and out and rotate to view different angles of the meshes.

For the Point Cloud Generator, the fields that the user can change as input for the Point Clouds are: the size of the Point Clouds, resolution, and model type.

2.3.1 Point Cloud Size

For the size of the Point Cloud, we believe that the best range for our project would be between 1000 and 10,000 vertices. We believe so because, we believe that Point Clouds of less than 1000 vertices would unlikely allow the Backend to properly reconstruct the meshes. Whilst, on the other end, inorder to obtain a better user experience, we believe that a Point Cloud of more than 10,000 vertices would have the Backend take too long to compute the reconstructed mesh.

2.3.2 Resolution

The user will be given 3 resolution options to render the mesh at: 64, 96, 128. This will determine the smoothness of the reconstructed object. We explain how resolution works in Section 5.

2.3.3 Model Type

The Model Type has 3 fields, named: Level1, Level2, Level3. They correspond to the levels of complexity of the three models we pretrained in the Backend.Explanations are provided in Section 4.1.

3 Methods: Middleware Connecting Interface and Pretrained Models

The functionality of this is done in Python, in the Flask framework. Most of the work is done in our app.py file. Here, for a given function, we create a unique request that the Frontend can access and as well as the Backend.

Requests that we have currently are:

1. From the FrontEnd, a request can be sent to retrieve the stored meshes in the Backend. The middleware then forwards this to the Backend and the Backend responds with the corresponding mesh and forwards it to the FrontEnd.
2. From the FrontEnd, a 'POST' request is sent to generate the Point Clouds. What passed on along with this request is an array of 30,000 vertices, an integer of the number of Point Cloud samples to create, and the type of Mesh. When the middleware receives this request, it forwards this to the Backend and the Backend responds with the corresponding reconstructed mesh, along with the Chamfer Distance metric and forwards it to the FrontEnd.

4 Methods: Backend – Training Signed Distance Functions

4.1 Problem Formulation

Signed Distance Functions We are interested in learning a continuous representation for 3D geometry. We learn a signed distance function (SDF), which is a continuous function that represents the surface of a shape approximated by a neural network [13, 14]. Given a raw input point cloud $P = \{p_i \in R^3\}_{i=1}^N$ with N points and an arbitrary number of 3D query points $x \in R^3$, we optimize a neural network Φ such that the network predicts the signed distance value of any given 3D point x to the shape described by the input point cloud. Then Φ is a unified function that can be directly applied to any target shape.

$$\Phi(x, P) = s, \tag{1}$$

where s denotes the predicted signed distance value of x.

The surface boundary of a shape described by P is its zero-level set $S_0(\Phi(P))$, which can be expressed as

$$S_0(\Phi(P)) = \{x \in R^3 \mid \Phi(x, P) = 0\}. \tag{2}$$

Thus, given a trained Φ, we can visualize the surface of some P by drawing its zero-level set.

Problem Setting We are given a dataset of meshes $X = \{P, SDF\}$, where P is a set of point clouds and SDF denotes the ground-truth SDF operator that is defined for all query points $x \in R^3$. Each $p_i \in P$ is a set of 3D coordinates that lie on the surface boundary of an object, and contains no additional information such as normals.

We train three models with varying levels of complexity. We denote them as M_{single} for "single object," M_{meta} for "meta-learning," and M_{prior} for "learning shape priors." In the user interface, they correspond to "Level1, Level2, Level3" respectively. The motivation and real-world applications for these three models are different. M_{single} has fast training and inference speeds, but each network is trained to overfit on one object. This can be useful for downstream tasks such as simulations (e.g. virtual reality, augmented reality, animations) where the same objects are repeatedly used. By using a small network, each object can have the same memory footprint as some standard graphics solutions but with higher accuracy [7].

M_{meta} uses the same lightweight network but trains with $O(nm)$ complexity where n is the number of meshes and m is the number of inner-loop iterations; i.e., iterations for calculating the second-order gradient, typically around 5 [14]. During inference, additional optimization and tuning on the target shape is also required. However, unlike M_{single}, M_{meta} is able to train on multiple meshes, increasing potential applications where some extra time during inference is permissible. This can be useful for downstream tasks such as medical imaging and analysis.

Finally, M_{prior} uses an auto-encoder framework, with the same 8 fully connected layers as the decoder and a PointNet as encoder [9, 16] for learning shape priors. Due to the substantial increase in the number of parameters of the encoder, both training and inference time is long. However, it is able to learn shapes even across multiple categories and can fit to any target data without additional tuning. By exploring acceleration structures such as octrees [10], it may be possible to achieve both generalization capabilities and efficiency; we leave this for future work.

4.2 M_{single}: Learning a Single Shape

Given a single object p_i, we sample K query points near the surface for training. Since we have the ground truth SDF operator; i.e., we know the signed distance values for all query points $x \in R^3$, the loss function is simply

$$\mathcal{L} = \frac{1}{K} \sum_{k \in K} \|\Phi(x_k) - SDF(x_k)\|_1 \tag{3}$$

where $\|\cdot\|_1$ is the L1 loss. We repeatedly train the query points with this objective until convergence.

4.3 M_{meta}: Meta-learning for Fast Adaptation

We follow the training scheme in [14]. Given a dataset with multiple meshes X, we train all meshes using the same architecture in M_{single}. However, for each training step of each mesh, we sample

half the points as "context" and half as "query." We compute second-order gradients on the "context" points for 5 iterations and use the generated network parameters to predict the "query" points. The final loss function is dependent on the prediction of the "query" points, so the model learns to quickly adapt to a given input in 5 iterations. For a more accurate and detailed depiction, see our Appendix, where we borrow the algorithm from [14]. When both adapting to the "context" points and predicting the "query" points, we use Equation 3 as before.

4.4 M_{prior}: Learning Shape Priors for Generalization

We use the PointNet from [16], which learns local and spatial features. However, to reduce model parameters, we do not follow their training step. After producing a shape feature z given an input point cloud, [16] uses a 5 layer network, where in each layer, they 1) concatenate z and a query point x and pass the resulting vector through a ResNet block and 2) pass z through a fully connected layer, and concatentate it to the output of 1). Instead, we simply concatenate z with x and pass the vector into the same 8 layer fully connected decoder. This reduces the number of parameters drastically and also allows for a clearer comparison to M_{single} and M_{meta}. For details on the encoder, refer to our Appendix.

5 Results

For all models, we train locally and deploy the pretrained models for inference. In this section, we show their experimental results. For data, we use ShapeNet [19], which has clean, synthetic 3D models with 270+ categories. We sample points on and near the surface for point cloud and query data, respectively. Since we have the mesh, it is easy to calculate ground truth signed distance values for supervised training. We mainly follow the sampling method and numbers from [13]. During each training step, we sample 1024 point cloud points and 16384 query points.

In our presentation and live demo, we will show results directly on our frontend, similar to Section 2 above. Here, for visual purposes we show the reconstructed results without additional distraction. In our interface, we provide 3 resolutions to select from: 64, 96, 128. This determines the size of the grid when running the Marching Cubes [20] algorithm; i.e., 64 means the unit cube is uniformly split into 64^3 points, and so the larger the number, the smoother the interpolation of the zero-level surface.

5.1 M_{single}

We use a single object from the QueenBed class in ShapeNet [19]. We show the ground truth and reconstruction results in different resolutions in Fig. Since this model is designed to overfit on one shape, users will also only be able to reconstruct this specific object with no adjustments except for resolution. In the figure below, we find that a resolution of 96 is a good balance between accuracy and time. The time took to reconstruct these meshes using a 2018 MacBook Pro (cpu only): 2 seconds, 4 seconds, and 7 seconds, respectively. This includes overhead such as loading the pretrained model, sampling points, and is almost identical to the time it takes for our frontend to process input and display these results.

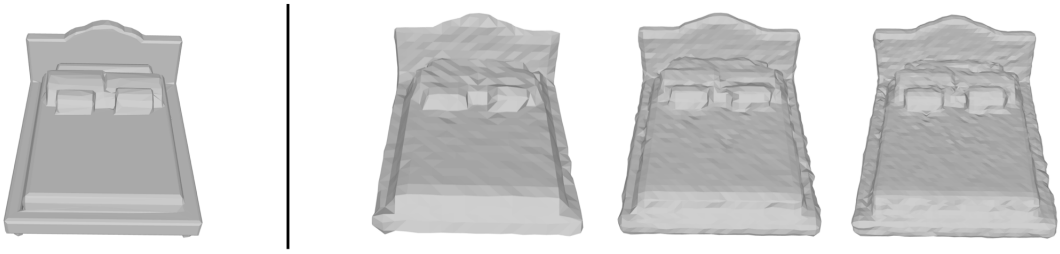


Figure 7: Training on testing on a single object. From left to right: ground truth object, reconstruction with resolution 64 (2 seconds), resolution 96 (4 seconds), resolution 128 (7 seconds).

5.2 M_{meta}

We use four objects from the same QueenBed class. We show some reconstruction results. Since we fine tune on the target shape during inference, users can apply mesh processing filters to verify the generalization capabilities of the model. As expected, reconstruction results are not as detailed as when overfitting to one shape, but the overall contour is generalized well. At the time of writing, we have not been able to thoroughly test their reconstruction time. However, since we use the same architecture as M_{single}, the time required should simply be multiplied by 5–10 seconds, 20 seconds, and 35 seconds, respectively for resolutions 64, 96, 128. Maybe would be faster due to caching and reduction of overhead.

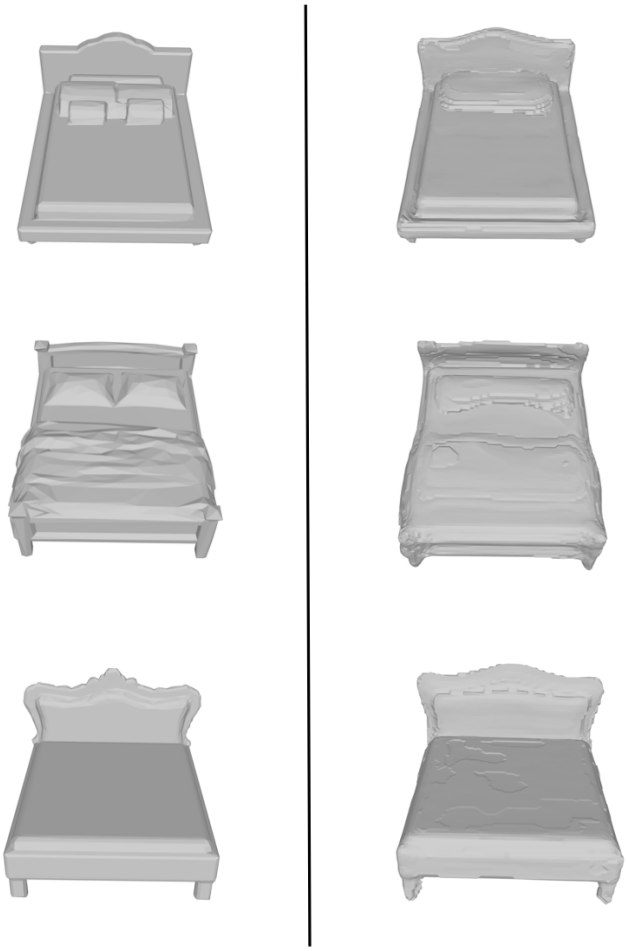


Figure 8: Training and testing on four objects of the QueenBed class. The meshes of the left of the line are the ground truth. The meshes on the right of the line are reconstructed results, all in resolution 128.

5.3 M_{prior}

We pretrain this model on 4 classes and 352 meshes in the ShapeNet dataset. We show some reconstruction results. Users are welcome to apply mesh processing filters as well as upload their own meshes to verify the generalization capabilities of the model. Even with no tuning on target data, accurate shape information generated by the encoder leads to favorable results. In this figure below, we further show results after fitting the encoder on the input point cloud for a few iterations. Unfortunately, in our interface we do not provide the option for this optimization due to time constraints.

Without tuning, the time to reconstruct for resolution 64 is 15 seconds including overhead. We have not thoroughly tested on other resolutions but we think this still is a reasonable time for user interaction.

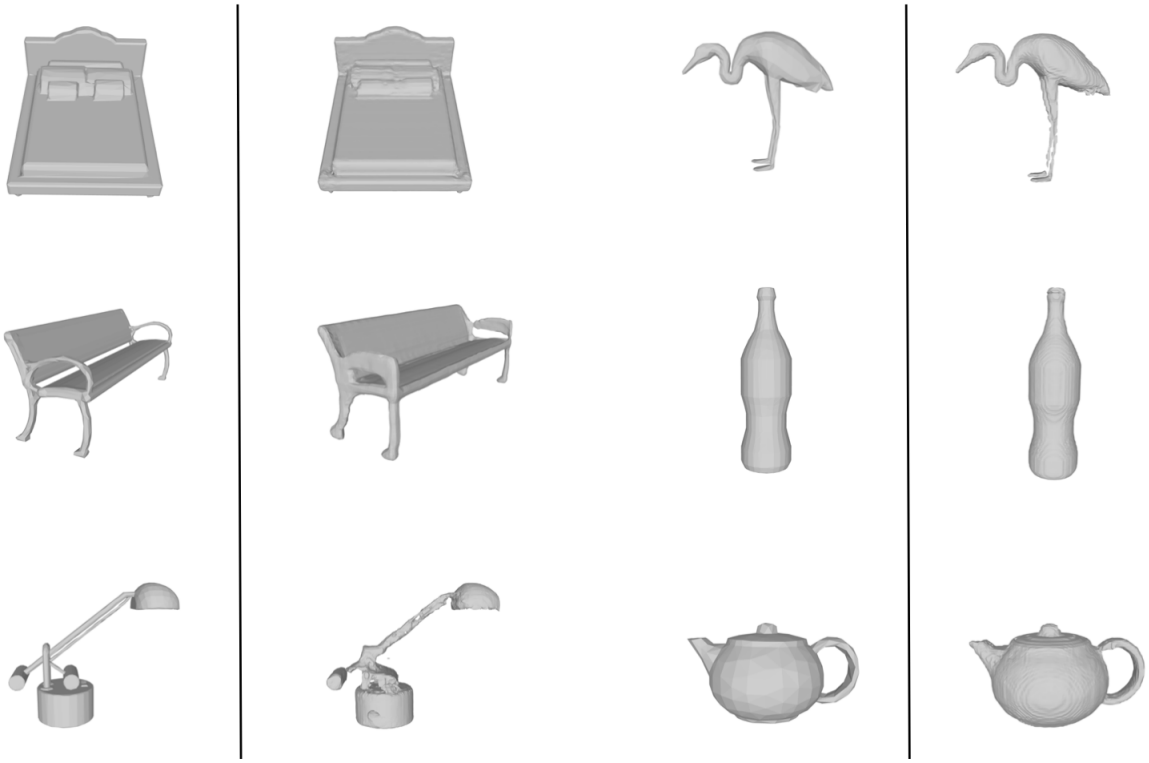


Figure 9: Training on 4 classes and 352 meshes. The first column contain results of reconstructing trained meshes without any tuning on target data. The second column contain results of reconstructing unseen classes with tuning on target data. With tuning, reconstructed results are very detailed and captures even subtle curvatures.

6 Conclusion

Each of our three models have certain tradeoffs that make them useful for varying applications. Our interface allows users to select appropriate parameters for exploring these tradeoffs in an accessible way.

Our main approach for creating lightweight models was through decreasing model parameters and dimensions, but the main bottleneck with a traditional SDF approach is that all points in a 3D grid need to be evaluated before a surface boundary can be drawn. In future work we would explore acceleration structures such as octrees [6, 10] so that empty spaces can be skipped.

7 Distribution of Workload

To accomplish this project we had to divide up the task among the three of us. For the Frontend, Herman was primarily assigned with implementing the Filters, adding helper functions, ensuring the meshes would render accordingly, and setting up the core Frontend logic and objects, could never have done it without him. Deus was involved in creating the Tutorial, ensuring the objects could be sent to the Backend appropriately and received by the Frontend, and ensuring that users could upload meshes. The middleware particularly gave us great difficulty throughout, Deus was primarily assigned to this and did great work in saving us. Deus also worked on stress-testing

different architectures locally and discussed with Gene about minimizing the network to attempt real-time inference. Gene, having the most experience with the Backend was primarily assigned to this. Being the core of our project, we don't know what we would have done without him. All of us in the end were involved in debugging, optimizing, and polishing the effects.

8 Honor Code

We pledge our honor that we did not violate the Honor Code during the completion of this project. – Gene Chou, Herman Ishengoma, Deus Nsenga

9 Acknowledgements

We would like to thank Prof. Felix Heide and Ethan Tseng for providing feedback on this project. We would particularly like to thank Ethan for suggesting methods for user interaction which made our project more accessible. We also don't know what we would have done without them.

References

[1] Matthew Berger, Andrea Tagliasacchi, Lee M. Seversky, Pierre Alliez, Gaël Guennebaud, Joshua A. Levine, Andrei Sharf, and Claudio T. Silva. A survey of surface reconstruction from point clouds. *Comput. Graph. Forum*, 36(1):301–329, jan 2017.

[2] Danny Driess, Jung-Su Ha, Marc Toussaint, and Russ Tedrake. Learning models as functionals of signed-distance fields for manipulation planning. In *5th Annual Conference on Robot Learning*, 2021.

[3] Pengfei Li, Yongliang Shi, Tianyu Liu, Hao Zhao, Guyue Zhou, and Ya-Qin Zhang. Semi-supervised implicit scene completion from sparse lidar, 2021.

[4] Roy Or-El, Xuan Luo, Mengyi Shan, Eli Shechtman, Jeong Joon Park, and Ira Kemelmacher-Shlizerman. Stylesdf: High-resolution 3d-consistent image and geometry generation. *arXiv e-prints*, pages arXiv–2112, 2021.

[5] David B Lindell, Dave Van Veen, Jeong Joon Park, and Gordon Wetzstein. Bacon: Band-limited coordinate networks for multiscale scene representation. *arXiv preprint arXiv:2112.04645*, 2021.

[6] Towaki Takikawa, Joey Litalien, Kangxue Yin, Karsten Kreis, Charles Loop, Derek Nowrouzezahrai, Alec Jacobson, Morgan McGuire, and Sanja Fidler. Neural geometric level of detail: Real-time rendering with implicit 3d shapes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11358–11367, 2021.

[7] Thomas Davies, Derek Nowrouzezahrai, and Alec Jacobson. On the effectiveness of weight-encoded neural implicit 3d shapes. *arXiv preprint arXiv:2009.09808*, 2020.

[8] Daniel Maturana and Sebastian Scherer. Voxnet: A 3d convolutional neural network for real-time object recognition. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 922–928, 2015.

[9] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 652–660, 2017.

[10] Maxim Tatarchenko, Alexey Dosovitskiy, and Thomas Brox. Octree generating networks: Efficient convolutional architectures for high-resolution 3d outputs. In *Proceedings of the IEEE international conference on computer vision*, pages 2088–2096, 2017.

[11] Lingjie Liu, Jiatao Gu, Kyaw Zaw Lin, Tat-Seng Chua, and Christian Theobalt. Neural sparse voxel fields. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 15651–15663. Curran Associates, Inc., 2020.

[12] Thibault Groueix, Matthew Fisher, Vladimir G. Kim, Bryan Russell, and Mathieu Aubry. AtlasNet: A Papier-Mâché Approach to Learning 3D Surface Generation. In *Proceedings IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[13] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 165–174, 2019.

[14] Vincent Sitzmann, Eric R. Chan, Richard Tucker, Noah Snavely, and Gordon Wetzstein. Metasdf: Meta-learning signed distance functions. In *Proc. NeurIPS*, 2020.

[15] Vincent Sitzmann, Julien N.P. Martel, Alexander W. Bergman, David B. Lindell, and Gordon Wetzstein. Implicit neural representations with periodic activation functions. In *arXiv*, 2020.

[16] Lars Mescheder Marc Pollefeys Andreas Geiger Songyou Peng, Michael Niemeyer. Convolutional occupancy networks. In *European Conference on Computer Vision (ECCV)*, 2020.

[17] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham, 2015. Springer International Publishing.

[18] Loading 3d models. `https://threejs.org/docs/#manual/en/introduction/Loading-3D-models`.

[19] Angel X Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, et al. Shapenet: An information-rich 3d model repository. *arXiv preprint arXiv:1512.03012*, 2015.

[20] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, page 163–169, New York, NY, USA, 1987. Association for Computing Machinery.

A Appendix

Here we borrow information from MetaSDF [14] and Convolutional Occupancy Networks [16] to show our training process in more detail. Unless noted otherwise, we follow their implementations. We do not take credit for these implementations.

A.1 Meta-learning Algorithm

Algorithm 1 MetaSDF: Gradient-based meta-learning of shape spaces

Precondition: Distribution $\mathcal{D}$ over SDF samples, outer learning rate β, number of inner-loop steps k

Initialize inner, per-parameter learning rates α, meta-parameters θ
while not done **do**
  Sample batch of shape datasets $X_i \sim \mathcal{D}$
  for all X_i **do**
    Split X_i into X_i^{train}, X_i^{test}
    Initialize $\phi_i^0 = \theta$, $\mathcal{L}_{train} = 0$
    for $j = 0$ **to** k **do**
      $\mathcal{L}_{train} \leftarrow \frac{1}{|X_i^{train}|} \sum_{(\mathbf{x},s)\in X_i^{train}} \ell_1\left(\Phi(\mathbf{x};\phi_i^j), s\right)$
      $\phi_i^{j+1} \leftarrow \phi_i^j - \alpha \odot \nabla_{\phi_i^j} \mathcal{L}_{train}$
    $\mathcal{L}_{test} \leftarrow \mathcal{L}_{test} + \frac{1}{|X_i^{test}|} \sum_{(\mathbf{x},s)\in X_i^{test}} \ell_1\left(\Phi(\mathbf{x};\phi_i^k), s\right)$
  $\theta, \alpha \leftarrow (\theta, \alpha) - \beta \nabla_{(\theta,\alpha)} \mathcal{L}_{test}$
return θ, α

A.2 PointNet Architecture

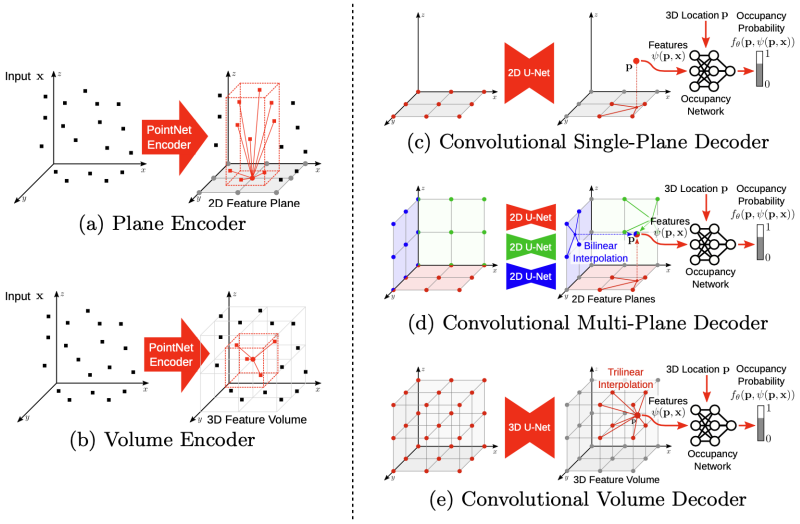


Fig. 2: **Model Overview.** The **encoder (left)** first converts the 3D input $\mathbf{x}$ (e.g., noisy point clouds or coarse voxel grids) into features using task-specific neural networks. Next, the features are projected onto one or multiple planes (Fig. 2a) or into a volume (Fig. 2b) using average pooling. The **convolutional decoder (right)** processes the resulting feature planes/volume using 2D/3D U-Nets to aggregate local and global information. For a query point $\mathbf{p} \in \mathbb{R}^3$, the point-wise feature vector $\psi(\mathbf{x}, \mathbf{p})$ is obtained via bilinear (Fig. 2c and Fig. 2d) or trilinear (Fig. 2e) interpolation. Given feature vector $\psi(\mathbf{x}, \mathbf{p})$ at location $\mathbf{p}$, the occupancy probability is predicted using a fully-connected network $f_\theta(\mathbf{p}, \psi(\mathbf{p}, \mathbf{x}))$.

In our experiments, we use (a) and (d) from the figure above. We modify how the shape features are concatenated to query points. Also, we do not use their occupancy decoder.