DEEP GENERATIVE SCULPTING MODELS FOR SINGLE IMAGE 3D RECONSTRUCTION

by

JASON JENNINGS

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON
December 2023

To my mother Sharon and my father Lawrence.

ACKNOWLEDGEMENTS

First, I would like to thank my professor Dr. Farhad Kamangar for his mentorship and guidance throughout my Ph.D. This would not be possible without his encouragement, kindness, understanding or knowledge.

Second, I would like to thank my committee members: Dr. Huber, Dr. Athitsos, and Dr. Zaruba for their support.

I would also like to thank the CSE Department at UTA, Dr. Ifshaq Ahmad and Dr. Zaruba for their support via the NSF GAANN program.

December 12, 2023

ABSTRACT

DEEP GENERATIVE SCULPTING MODELS FOR SINGLE IMAGE 3D RECONSTRUCTION

JASON JENNINGS, Ph.D.

The University of Texas at Arlington, 2023

Supervising Professor: Farhad Kamangar

In the field of computer vision, learning representations of images is an important task. This dissertation introduces deep generative sculpting models (DGSM), deep learning models that learn 3D representations of objects from 2D images. DGSMs use convolutional networks combined with a differentiable renderer to attempt to "sculpt" a base 3D mesh, such as a sphere, to faithfully represent an object in the scene, and render it to reconstruct the input image.

The core methodology revolves around the encoding of the input image into latent variables. These variables are decoded into interpretable scene parameters, describing the object's translation, rotation, scale, texture, and "sculpting parameters". These are used to build a scene, and render it using a differentiable renderer.

Because DGSMs use a differentiable renderer, all of the latent variables describing an image are mapped directly to a parameter a human can understand, such as a scale factor, translation vector, rotation angle, or adjustment to a vertex of a triangle mesh.

In this dissertation we investigate two different models: The additive model, wherein each vertex undergoes independent adjustments. The warping model, characterized by a single-shot transformation using Gaussian Radial Basis Functions (RBFs).

We perform experiments on synthetic data rendered from 3D models. Our focus in this work is datasets that contain a single class of object. Our synthetic data consists of three datasets: faces, cars, and airplanes.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

INTRODUCTION AND BACKGROUND

1.1 Background

This section serves as a reference to some of the fundamental building blocks of the models and concepts used in this dissertation. Every model built and trained in this dissertation is in large part composed of different aspects of these fundamental models, along with some novel components.

1.2 Autoencoders and Generative Models

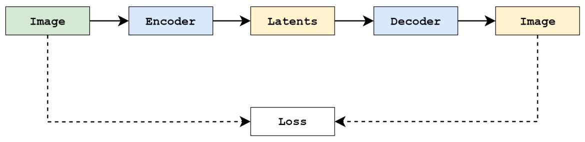

Figure 1.1. A block diagram of a basic autoencoder model.

An auto-encoder is a neural network architecture wherein the input is fed into an ”encoder” network, mapped to a set of ”latent variables” and the latent variables are then fed into a decoder network, which attempts to reconstruct the original input.

Autoencoders make use of some limitation of capacity, such as mapping to a lower dimensional space than the original inputs, to avoid trivial solutions.

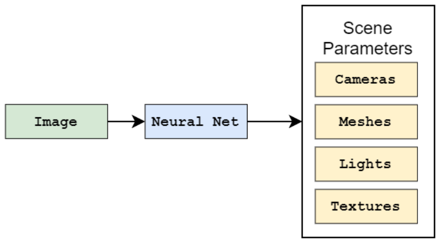

Figure 1.2. A conceptual model of an inverse graphics network.

1.3 Inverse Graphics

Inverse graphics is a computer vision concept. Concretely, an inverse graphics approach poses solving specific computer vision problems as discovering the ”graphic codes” or scene parameters of interest related to the scene in a natural image.

1.4 Differentiable Renderering

A differentiable renderer is a special renderer with two functions: forward and backward. In the forward pass (see: 1.3 the renderer generates an image of the input scene. This is no different from any other renderer. In the backward pass (see: 1.3, gradients with respect to the image (or some function of the image) flow backward to the scene parameters, allowing us to compute partial derivatives with respect to them, and perform gradient based optimization.

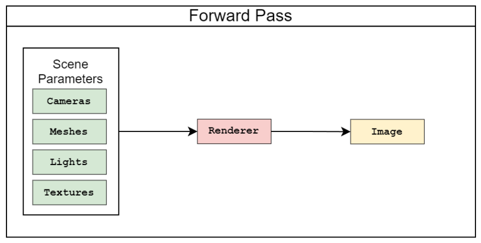

Figure 1.3. In the forward pass, a differentiable renderer is conceptually no different from any other renderer. It generates an image of the input scene..

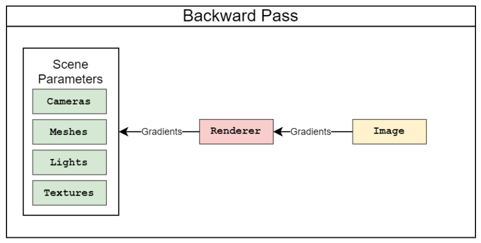

Figure 1.4. The backward pass of a differentiable renderer. Gradients flow back from the image to the various scene parameters..

CHAPTER 2

The Additive Sculpting Model

2.1 Model Description

Our first sculpting model, we call the "Additive" Sculpting model. Our image is input into an Encoder network, mapped to a D dimensional latent vector, which is input into the Texture Decoder, Vertex Offset Decoder, and Scene Parameter Decoders, to construct our scene, and then rendered by our differentiable renderer.

In our later models, much of the architecture remains the same. In essence, our "sculpting models" focus on different methods of generating vertex offsets.

2.1.1 Encoder

For our encoder, rather than starting from random weights, we utilize a pre-trained ResNet-18 [1] from the pytorch [2] library, and fine-tune it for our task. To it we attach a single linear layer to map the output of the "FC" layer to our D dimensional latent vector.

2.1.2 Texture Decoder

Our texture decoder is implemented using ConvTranspose2D layers. The basic architecture was popularized in DCGAN [3]. First we define a **Block**(I, N, K) as a sequence of layers: ConvTranspose2D layer (with I input channels, N convolutions with kernel side K) followed by a batch normalization layer, and a LeakyReLU activation function (see figure 2.1).

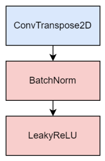

Figure 2.1. One block of our convolutional network consists of a ConvTranspose2D layer, followed by a batch normalization layer, and a LeakyReLU activation function.

We can then define our texture decoder network in terms of blocks, as seen in 2.2.

We use a Sigmoid activation function in the final layer to make sure our texture's values are valid inputs into our renderer, which requires texture pixel values to be 32 bit floats between 0.0 and 1.0.

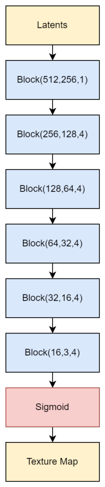

Figure 2.2. Network architecture of texture decoder.

2.1.3 Vertex Offset Decoder

Our Vertex Offset Decoder is a simple Linear layer with Nx3 nodes (where N is the number of vertices in the base mesh), that accepts our D dimensional latent vector, and maps it to a [N, 3] matrix of translation vectors.

2.1.4 Scene Parameter Decoders

We describe the details of the remaining scene parameter decoders below.

2.1.4.1 Translation Decoder

A simple Linear layer with 3 nodes, mapping the latents to a translation vector: $\mathbf{t} = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$. This allows our model to learn where to center our object in the scene.

2.1.4.2 Rotation Decoder

The rotation decoder is also a simple Linear layer with 3 nodes, mapping the latents to euler angles (γ, β, α) These angles are then used to build a composite transformation matrix Concretely, we construct the matrices: $R_x(\gamma) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{bmatrix}$

$$\mathrm{R}_y(\beta) = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix}$$

$$\mathrm{R}_z(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

We can then create our composite rotation matrix, by composing the 3 matrices: $R = R_x(\gamma)R_y(\beta)R_z(\alpha)$

2.1.4.3 Scale Decoder

Also simple Linear layer with 3 nodes, mapping the latents to a scale factors s_x, s_y, s_z. This allows our model to influence the overall size of the object without needing to adjust all of the vertices individually.

For this layer, we use the ReLU activation function, to prevent negative scale values. Additionally, to prevent the model from shrinking the base mesh too much at initialization, we initialize the weights with parameters and bias, such that the default scale values center around 1.0.

2.1.5 Differentiable Renderer

Though our work makes use of a differentiable renderer, we do not build a novel differentiable renderer. Instead, we use the wonderful differentiable renderer from PyTorch3D [4]. We discuss the capabilities of PyTorch3D in 2.1.5.1

2.1.5.1 Renderer Details

The PyTorch3D differentiable renderer has two main functions:

1. Rasterization - Mesh objects are transformed from world coordinates to camera coordinates, and are then rasterized into "Fragments", which contain the following information for each pixel in the output image: which faces intersect with that pixel, z-buffer (depth) of the intersection point of those faces, barycentric coordinates of the intersection points, and other information that may be useful for the rendering process.

2. Shading - The shader modules take in the output of the rasterizer, as well as the meshes and scene parameters, and computes the final color of each pixel.

   The Pytorch3D differentiable renderer support texture mapped meshes, a single light per scene (either ambient, directional, or spotlight), and several classic shading models such as Goraud shading and Phong Shading.

   In our experiments, we use Soft Phong Shading and a single ambient light with fixed parameters.

2.2 Laplacian Regularization

Initial experiments using mean squared error often resulted in a very jagged or bumpy mesh, so we use Laplacian Regularization as used in [5] to encourage a smooth mesh.

Given a mesh with vertices V and a function $f : V \to \mathbb{R}^3$ mapping each vertex to its 3D position, the Laplacian at a vertex v_i is defined as a weighted sum over the differences between $f(v_i)$ and the positions of its neighboring vertices $f(v_j)$.

The cotangent weights are calculated using the opposite angles α and β of the edge connecting v_i and v_j. Specifically, the weight w_{ij} for the edge is:

$$w_{ij} = \cot(\alpha) + \cot(\beta) \tag{2.1}$$

Thus, the Laplacian regularization term, which encourages mesh smoothness, can be formulated as:

$$L_{\text{smooth}}(f) = \frac{1}{2} \sum_{(i,j)\in\text{edges}} w_{ij} \|f(v_i) - f(v_j)\|^2 \tag{2.2}$$

2.3 Loss Function

Given an input image x and its reconstructed version $\hat{x}$, and their corresponding silhouettes S_x and $S_{\hat{x}}$, the loss function $\mathcal{L}$ comprises three main components: $L_{\text{mse}}, L_{\text{mask}}$, and L_{smooth}.

$$L_{\text{mse}}(A, B) = \frac{1}{N} \sum_{i=1}^{N} (a_i - b_i)^2 \tag{2.3}$$

$$L_{\text{mask}}(A, B) = \frac{1}{N} \sum_{i=1}^{N} (a_i - b_i)^2 \tag{2.4}$$

$$\tag{2.5}$$

2.4 Total System Diagram

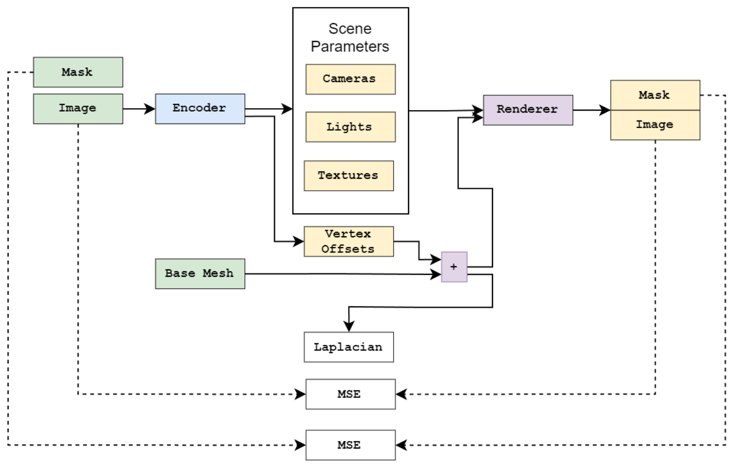

Figure 2.3. A block diagram of a the additive sculpting model model.

2.5 Performance Metrics

In order to quantitatively evaluate the performance of our models, we make use of three performance metrics: Mean Squared Error, Voxelized Intersection over Union, and Chamfer Distance.

2.5.1 Image Reconstruction Error (Mean Squared Error)

Our first performance metric is our reconstruction error (MSE). This measures how well our model does at reconstructing the target image. It is comparable across different classes of models, including models that do not make use of 3D models or differentiable rendering.

2.5.2 Chamfer Distance

The Chamfer distance has been used historically in both computer vision and computer graphics for measuring the dissimilarity between two sets of points, commonly point clouds. Chamfer distance calculates the average distance of each point in one set to its nearest point in the other set. Within the context of deep learning, Chamfer distance was popularized by its use as a loss function in PointNet [6]. Formally:

$$\begin{aligned}&\text{Given two point sets } A = \{a_1, a_2, ..., a_m\} \text{ and } B = \{b_1, b_2, ..., b_n\},\\&\text{the Chamfer distance CD}(A, B) \text{ is defined as:}\\&CD(A,B) = \frac{1}{m}\sum_{a\in A}\min_{b\in B}||a-b||_2^2 + \frac{1}{n}\sum_{b\in B}\min_{a\in A}||a-b||_2^2\end{aligned} \tag{2.5}$$

In this work we do not use 3D supervision, so we cannot optimize Chamfer Distance directly. Instead, we use it after training to measure our model's performance on the 3D reconstruction task.

Concretely we sample m points on the surface of our reconstructed mesh, and m points on the surface of our ground truth mesh. The pointclouds are then normalized by shifting each of their means to 0, and having a maximum extent of 1, and then the Chamfer distance is calculated as described in 2.5.2.

We primarily use Chamfer distance for our face model. The Basel face model is not a water-tight mesh, so we cannot easily convert it to a volumetric representation for comparison. The Chamfer distance allows us to quantify our performance using points sampled from the surface, rather than a volumetric representation.

2.6 Modeling Faces with a Sphere

We train our additive model on our face dataset with the following parameters:

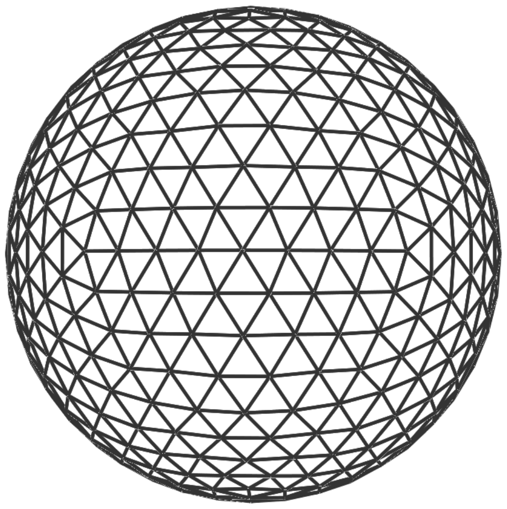

Figure 2.4. Our base sphere mesh with 3840 vertices and 1280 faces..

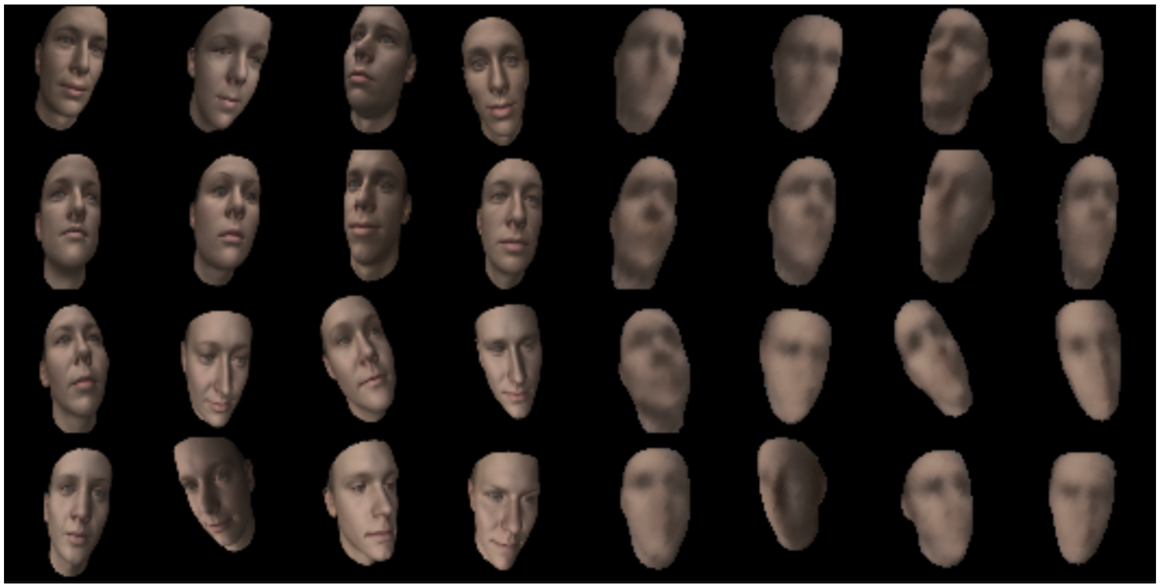

Figure 2.5. Ground truth faces (left grid), reconstructions (right grid).

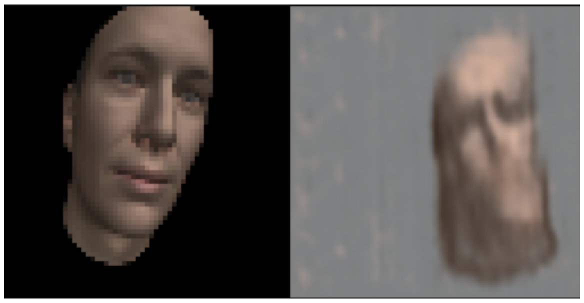

Figure 2.6. Illustration of the texture map for our sphere model. Left: Target Image, Right: Texture Map.

2.6.1 Analysis

The model learns to represent faces by mapping them onto the surface of a sphere, and even seems to make use of in-plane rotations in the data set, however the model fails that it would be beneficial to use the combination of surface geometry + out of plane rotations, instead relying on the texture network to do most of the heavy lifting. Since it's only using one small portion of the sphere to represent the face, most of the capacity of the texture decoder is wasted, and the performance suffers.

2.7 Modeling Cars with a Sphere

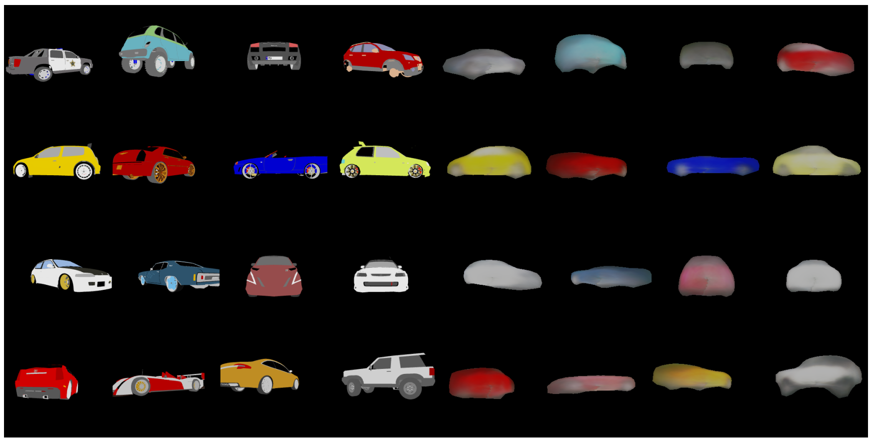

Figure 2.7. Left 4x4 grid (ground truth images), Right 4x4 grid (our model's reconstructions).

We repeat the previous experiment using our cars data set. See 2.7 for qualitative results.

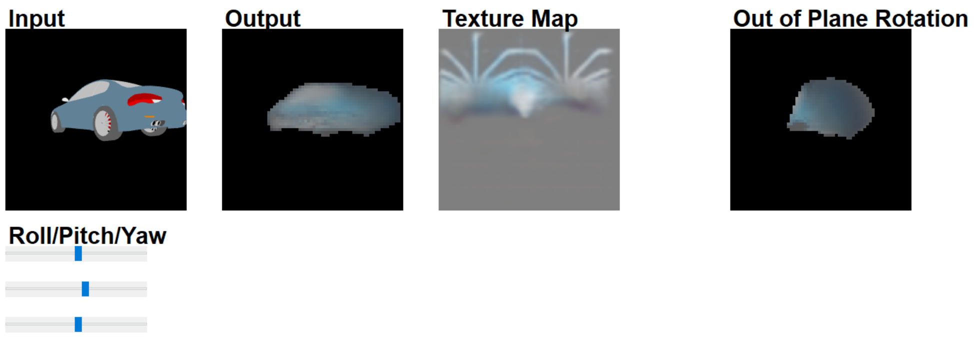

Figure 2.8. Visualizing the texture map and euler angles for a given image..

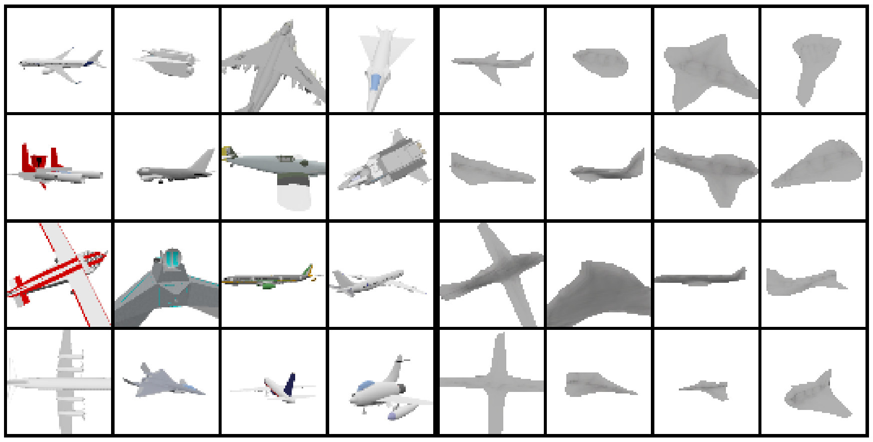

Figure 2.9. Left 4x4 grid (ground truth images), Right 4x4 grid (our model's reconstructions).

2.8 Modeling Planes with a Sphere

For our final experiment, we repeat the previous experiment on the planes data set.

See figure 2.9 for qualitative results. The quantitative results will be tabulated in the next chapter.

CHAPTER 3

The Warping Sculpting Model

3.1 Motivation

Our motivation in this work was to build a neural network model that reconstructs an input image by building ”sculpting” it, inspired by how 3D modelers build objects in tools such as Blender and ZBrush. In this section, we build upon the additive model, by building differentiable sculpting tools that operate more similarly to sculpting tools commonly used in 3D modeling programs.

Concretely, rather than our neural network adjusting each vertex of a triangle mesh independently, we would like to apply a set of ”sculpting” operations, by selecting points on or near the surface of the mesh to manipulate, and the parameters defining how the vertices in that area should be manipulated.

To accomplish this, we choose to use Radial Basis Functions (or RBFs) as our differentiable sculpting model.

3.2 Differentiable Sculpting Tool

3.2.1 Radial Basis Functions

Radial Basis Functions (RBFs) are a type of function whose value depends on the distance from a certain point, usually called a ”center”. They're commonly used in interpolation and neural networks, among other applications. A simple RBF can be defined as:

$$\phi(\mathbf{r}) = \phi(\|\mathbf{x} - \mathbf{c}\|)$$

Where:

- **x** is the input vector.
- **c** is the center of the RBF.
- $\| \cdot \|$ denotes the Euclidean norm.
- ϕ is the radial basis function, and $\mathbf{r}$ is the Euclidean distance between **x** and **c**.

Several types of RBFs exist, a few of which include:

1. Gaussian:
$$\phi(r) = e^{-\alpha r^2}$$
2. Multiquadric:
$$\phi(r) = \sqrt{r^2 + \alpha^2}$$
3. Inverse multiquadric:
$$\phi(r) = \frac{1}{\sqrt{r^2 + \alpha^2}}$$
4. Thin plate spline:
$$\phi(r) = r^2 \log(r)$$

Where α is a shape parameter that might control the spread or the shape of the RBF.

We investigate a RBF based sculpting model for a few reasons:

1. Locality - Local features have proven to be valuable for many computer vision tasks. Because RBFs are based around distance from a 'center' or keypoint, they are inherently capable of representing local features.
2. Flexibility - Though RBFs can naturally represent local features, they are also capable of representing global features, by adjusting their parameters (for example, by making α very small).
3. Dynamic Meshes - Unlike the additive sculpting model, which is directly tied to the number of vertices in the base mesh, the use of RBFs would decouple

the the neural network architecture from the number of vertices, and instead tie it to the number of keypoints/RBFs. This could potentially allow for the number of polygons in our mesh to be adjusted during training dynamically (i.e. polygons that grow too large could be subdivided, planar regions could be collapsed).

4. Smoothness - The additive sculpting model requires the use of regularization to prevent a jagged mesh. RBFs should naturally produce smooth deformations in the surface
5. Similarity to real world sculpting tools - We observe that radial basis functions when applied to a 3D mesh resemble sculpting brushes in Blender [7], specifically the "Inflate/Deflate" and "Blob" brush.

3.2.2 Sculpting Equation

In this work, we choose to focus on the use of Gaussian RBFs, with the following definition for our differentiable sculpting tool:

Given a set of n keypoints w_j with their corresponding warp vectors v_j, warp strength s, and a brush size k, the warped position $\hat{x_i}$ of a vertex x_i is given by:

$$\hat{x_i} = x_i + s \sum_{j=1}^{n} v_j e^{-k\|x_i - w_j\|^2} \tag{3.1}$$

3.2.3 Illustration of Warping Effect

Before integrating our differentiable sculpting tool into a neural network architecture, we attempt to qualitatively validate validate its ability to perform sculpting tasks.

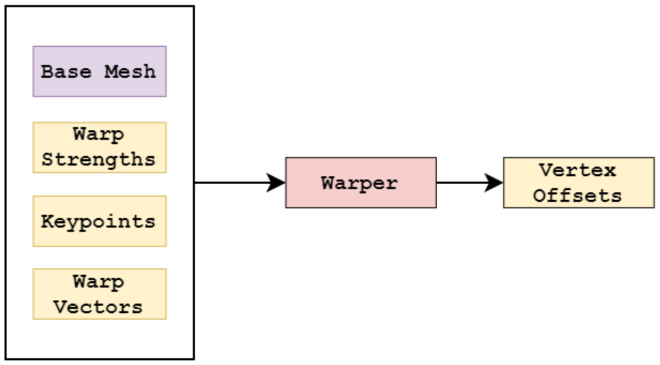

Figure 3.1. A block diagram of our RBF sculpting module..

3.2.3.1 Warping Teapots

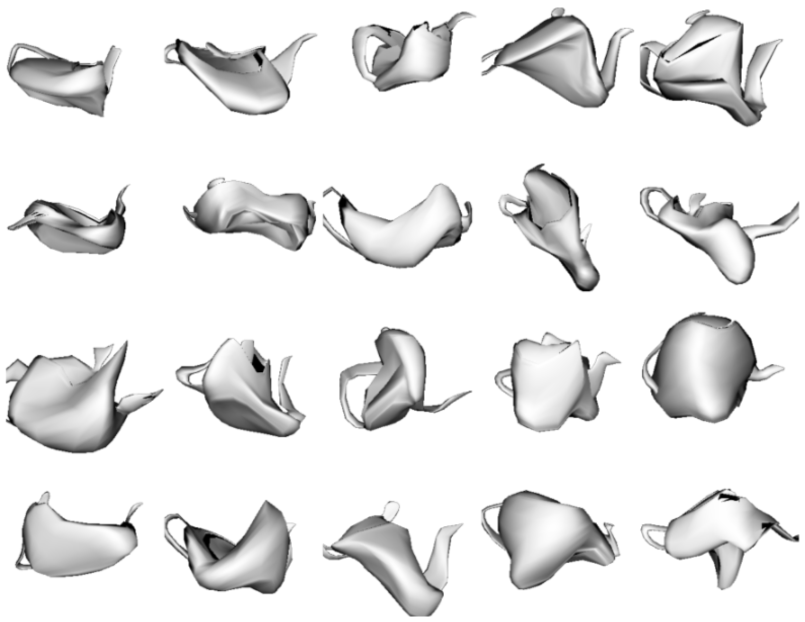

Figure 3.2. A 5x5 grid showing the effects of random warps on the Utah teapot.

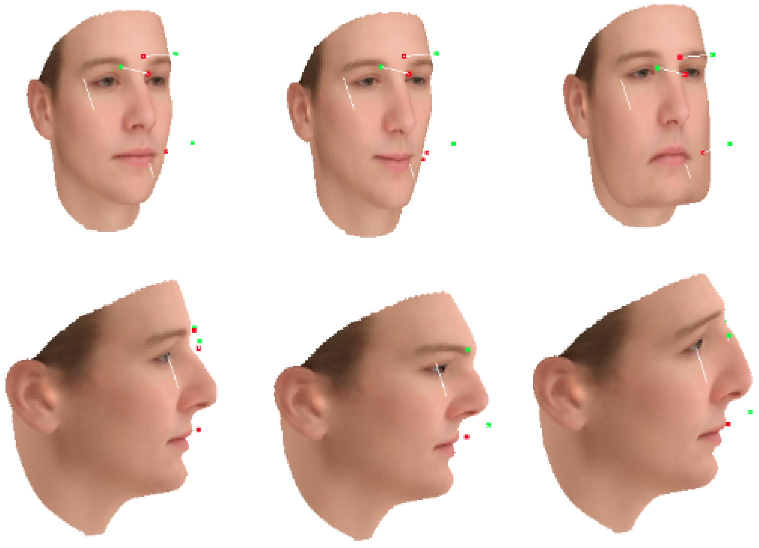

Figure 3.3. A 3x2 Grid demonstrating the ability of our differentiable sculpting model on human faces..

3.2.3.2 Warping Faces

Additionally, using the Basel face model [8], we designed a small user interface to test the effectiveness of our differentiable sculpting module at manipulating faces. Keypoints can be selected, and the strength adjusted.

Additionally, we applied our warps symmetrically, as it's a common feature in sculpting programs.

In 3.3 we can see with just a few symmetrically warps, we can produce plausible changes in the structure of the face model, such as widening or narrowing the jaw, increasing the size of the nose, the shape of the brow, and even changing a neutral expression to a smile or frown.

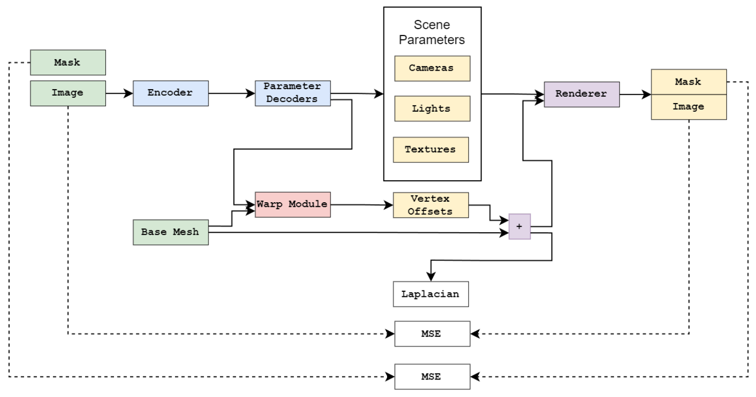

Figure 3.4. A block diagram of our full RBF sculpting model.

3.3 System Diagram

In figure 3.4 you can see the total system diagram of our RBF warping based sculpting model. If you refer back to the additive model in 2.3, you can see the warping model is virtually identical, but instead of using a Linear layer to calculate the vertex offsets, we use our Warping module 3.1.

3.4 Experiments

We train each our Additive Model and RBF Warping Model on each of our data sets.

We use the Adam [9] optimizer with learning rates: texture-decoder: 0.0003, base-vertices: 0.001, all other decoders: 0.00002

We use $L_{mse} = 1.0$ and $L_{mask} = 0.2$ for our loss coefficients

For all warping models, we use 256 warps, with keypoints initialized to points sampled from the surface of the base mesh, with a pertubation by a vector of length 0.1.

3.4.1 Modeling Faces with a Sphere

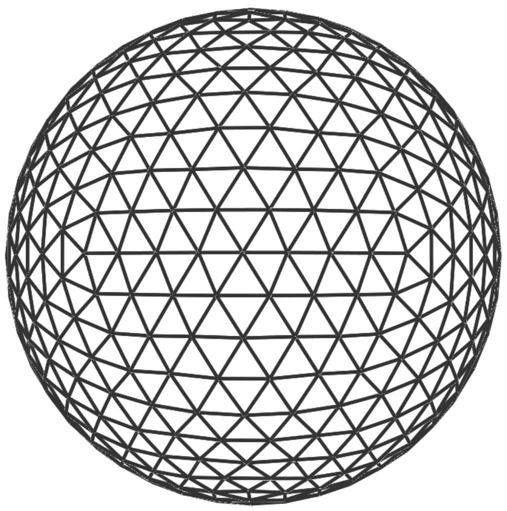

Figure 3.5. We utilize the same base mesh as in the previous section.

3.4.2 Modeling Cars with a Sphere

We repeat the previous experiment using our cars data set.

3.4.3 Modeling Planes with a Sphere

For our final experiment, we repeat the previous experiment on the planes data set. See

3.5 Quantitative Results

To put our results into context, we also trained a baseline model on each dataset. The model is a simple auto-encoder, as in 1.1.

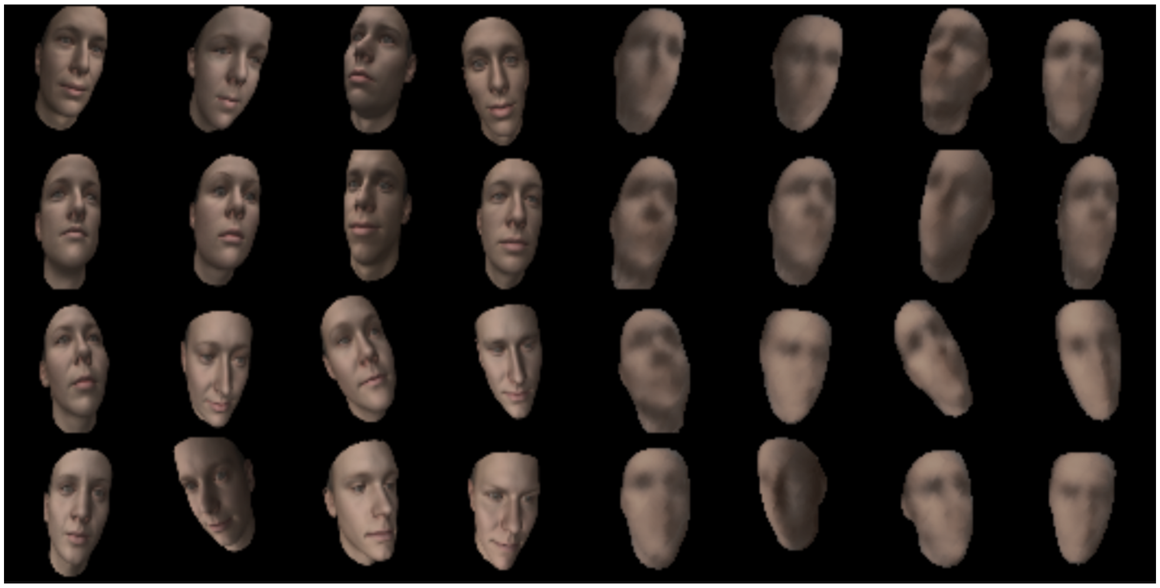

Figure 3.6. Ground truth faces (left grid), reconstructions (right grid).

The encoder network is a pretrained Resnet-18 [1] as in the Additive and RBF Sculpting models.

The decoder network is the same as the texture decoder in the Additive and RBF sculpting models.

	Cars	Planes	Faces
Baseline	0.00640	0.00381	0.00187
Additive	0.00754	0.00427	0.00291
RBF Warping	0.00796	0.00415	0.00286

Table 3.1. Validation mean squared error on each dataset, separated by model.

3.6 Conclusions

We have successfully implemented two sculpting models, the Additive model, and the RBF Warping based sculpting model. Though neither sculpting model beats

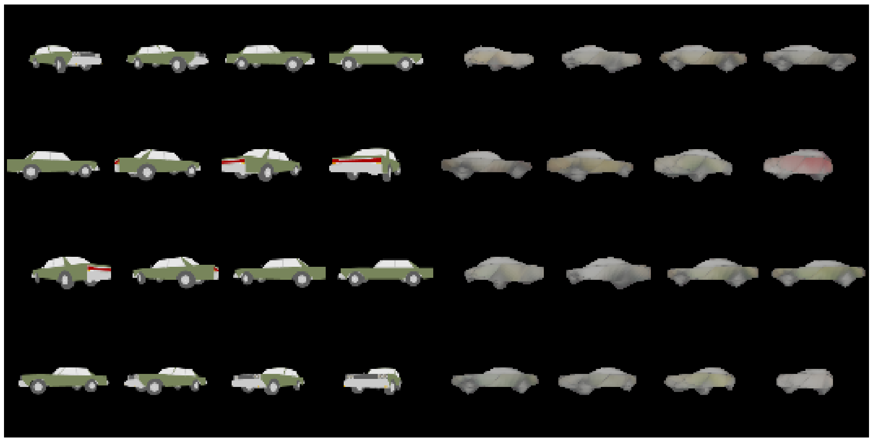

Figure 3.7. Left 4x4 grid (ground truth images), Right 4x4 grid (our model's reconstructions).

the baseline of a convolutional auto-encoder, it successfully learns the basic silhouette and color features of the input image. The model fails to construct a complete 3D model of the object, including parts not in the current view. This is not surprising, as this model does not make use of any multiview supervision. We leave the task of full 3D reconstructions including out-of-view features to later work. We believe this represents a successful first step towards deep learning models that learn

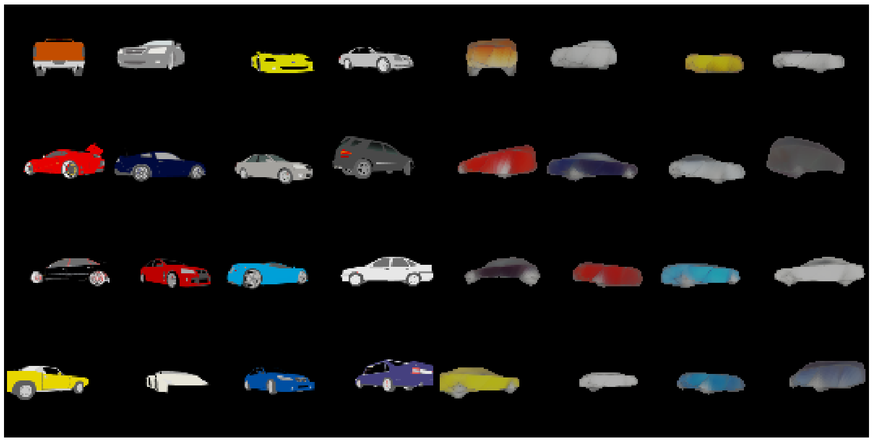

Figure 3.8. Left 4x4 grid (ground truth images), Right 4x4 grid (our model's reconstructions).

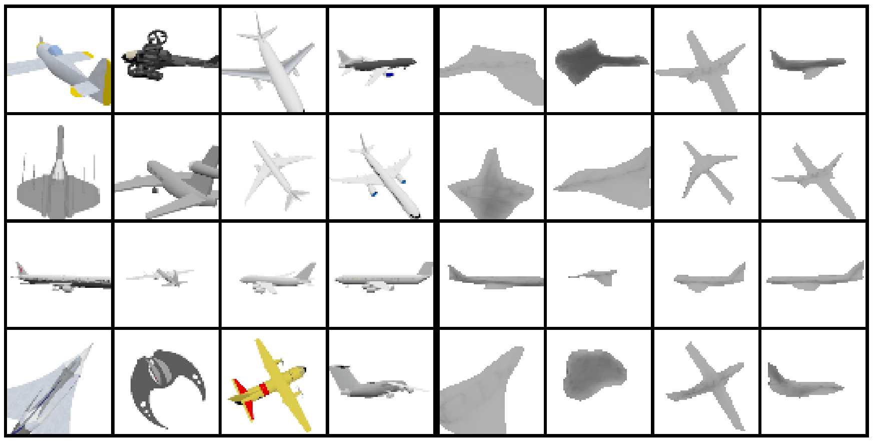

Figure 3.9. Left 4x4 grid (ground truth images), Right 4x4 grid (our model's reconstructions).

APPENDIX A

Code - Differentiable Renderer

A.0.1 Additive Model

A.0.1.1 Texture Decoder

```
import torch
from torch import nn

class TextureDecoder(nn.Module):
    def __init__(self, num_latents=512, base_convs=256):
        super(TextureDecoder, self).__init__()

        self.main = nn.Sequential(
            nn.ConvTranspose2d(num_latents, base_convs, 4, 1,
               0, bias=False),
            nn.BatchNorm2d(base_convs),
            nn.LeakyReLU(0.2, inplace=True),


            nn.ConvTranspose2d(base_convs, base_convs//2, 4,
               2, 1, bias=False),
            nn.BatchNorm2d(base_convs//2),
            nn.LeakyReLU(0.2, inplace=True),


            nn.ConvTranspose2d(base_convs//2, base_convs//4,
               4, 2, 1, bias=False),
            nn.BatchNorm2d(base_convs//4),
            nn.LeakyReLU(0.2, inplace=True),


            nn.ConvTranspose2d(base_convs//4, base_convs//8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(base_convs//8),
            nn.LeakyReLU(0.2, inplace=True),


            nn.ConvTranspose2d(base_convs//8, base_convs//16, 4, 2, 1, bias=False),
            nn.BatchNorm2d(base_convs//16),
            nn.LeakyReLU(0.2, inplace=True),


            nn.ConvTranspose2d(base_convs//16, base_convs//32, 4, 2, 1, bias=False),
            nn.BatchNorm2d(base_convs//32),
            nn.LeakyReLU(0.2, inplace=True),


            nn.ConvTranspose2d(base_convs//32, 3, 4, 2, 1, bias=False),

            nn.Sigmoid()
        )

    def forward(self, input):
        # Reshape the input tensor to (batch_size, latent_size
            , 1, 1)
        batch_size = input.shape[0]
        input = input.view(batch_size, -1, 1, 1)
        return self.main(input).permute(0, 2, 3, 1)
```

A.0.1.2 Full Model

```
import torch

class AdditiveModel(torch.nn.Module):
    def __init__(self, num_latents, verts, faces, uvs,
        uv_faces):
        # ... code omitted for brevity)
        self.encoder = ResnetEncoder(num_latents * 3)
        self.texture_decoder = TextureDecoder()
        self.position_decoder = torch.nn.Sequential(torch.nn.
            Linear(num_latents, 3))
        self.rotation_decoder = torch.nn.Sequential(torch.nn.
            Linear(num_latents, 3))
        self.scale_decoder = torch.nn.Sequential(torch.nn.
            Linear(num_latents, 3))

        self.renderer = Renderer()

        self._base_verts = torch.nn.Parameter(torch.tensor(
            verts, dtype=torch.float32))
```

```python
        self._base_faces = torch.tensor(faces, dtype=torch.int64)
        self._uv_faces = torch.tensor(uv_faces, dtype=torch.int64)
        self._base_uvs = torch.nn.Parameter(torch.tensor(uvs, dtype=torch.float32))

        # (... code omitted for brevity)

    def forward(self, x):
        batch_size = x.shape[0]
        latents = self.encoder(x)
        translation = self.position_decoder(latents).unsqueeze(1)
        angles = self.rotation_decoder(latents)
        vert_offsets = self.shape_decoder(latents).reshape(batch_size, -1, 3)
        scale_values = F.relu(self.scale_decoder(latents)) + 1.0
        texture_maps = self.texture_decoder(latents)
        rotation_matrix = EulerRotation(angles)

        adjusted_verts = self._base_verts * self.scale_values.unsqueeze(1) + vert_offsets
        adjusted_verts = adjusted_verts @ rotation_matrix.permute(0, 2, 1) + translation
        out = self.renderer(adjusted_verts, self._base_faces, self._base_uvs, texture_maps, self._uv_faces)
        silhouette = self.renderer(adjusted_verts, self._base_faces, self._base_uvs, texture_maps, self._uv_faces, silhouette=True)
        return [out, silhouette]
```

A.0.1.3 RBF Warper

```
import torch

class Warper(torch.nn.Module):
    def __init__(self, num_warps, keypoints):
        super().__init__()
        self._keypoints = torch.tensor(keypoints, requires_grad=True, dtype=torch.float32)
        self._wvs = torch.randn(num_warps, 3, requires_grad=True)
        self.register_buffer('keypoints', self._keypoints)
        self.register_buffer('wvs', self._wvs)
        self.last_warps = None
        self.alpha = 15

    def gaussian(self, D, falloff=1):
        inside = -D ** 2
        return -torch.exp(self.apha*inside)

```

```python
    def compute_warps(self, inputs, geometry, keypoints):
        D = torch.cdist(geometry, keypoints)
        DD = self.gaussian(D)
        out = torch.einsum('bij,bj,jd->bid', DD, inputs, self.wvs)
        return out

    def forward(self, inputs, geometry, falloffs=None):
        warps = self.compute_warps(inputs, geometry, self.keypoints, self.wvs, falloffs)
        self.last_warps = warps
        return geometry + warps
```

APPENDIX B

Dataset - Faces

B.1 Introduction

Because of the wealth of research into faces in computer vision, we were motivated to try our sculpting models on faces. Due to the complexity of modeling natural images with complex backgrounds, we decided to use synthetic data where we can control all the parameters of the scene. To do this, we make use of The Basel Face model.

B.2 The Basel Face Model

B.3 Sample Set 1

APPENDIX C

Dataset - Cars

C.1 Introduction

In addition to our novel face dataset, we synthesize a dataset of renderings of cars from the shapenet cars synset. We choose this dataset because cars are typically (roughly) a topological sphere, making them a good candidate for our model. Additionally, they demonstrate different kinds of variability (such as texture and color) that is not present in the cars dataset.

C.2 ShapeNet Cars

We synthesize each model in the shapenet cars synset from the same 20 views, rotated around its y axis in equal increments (see C.3.1). Additionally, to demonstrate the variety of cars represented in the dataset, see C.3.2.

Our dataset consists of 3535 3D models of cars, each rendered in the 20 viewpoints mentioned above for a total of 70700 samples. These are split into train, validation and test sets. We partition these samples into train, validation, and test sets. If a model is used in one of the sets, all 20 viewpoints are in that set.

Table C.1. Data Splits for Synthetic Shapenet Cars Dataset

Train	Val	Test
53960	8540	7480

C.3 Samples

C.3.1 Varying Rotation

C.3.2 Random Samples

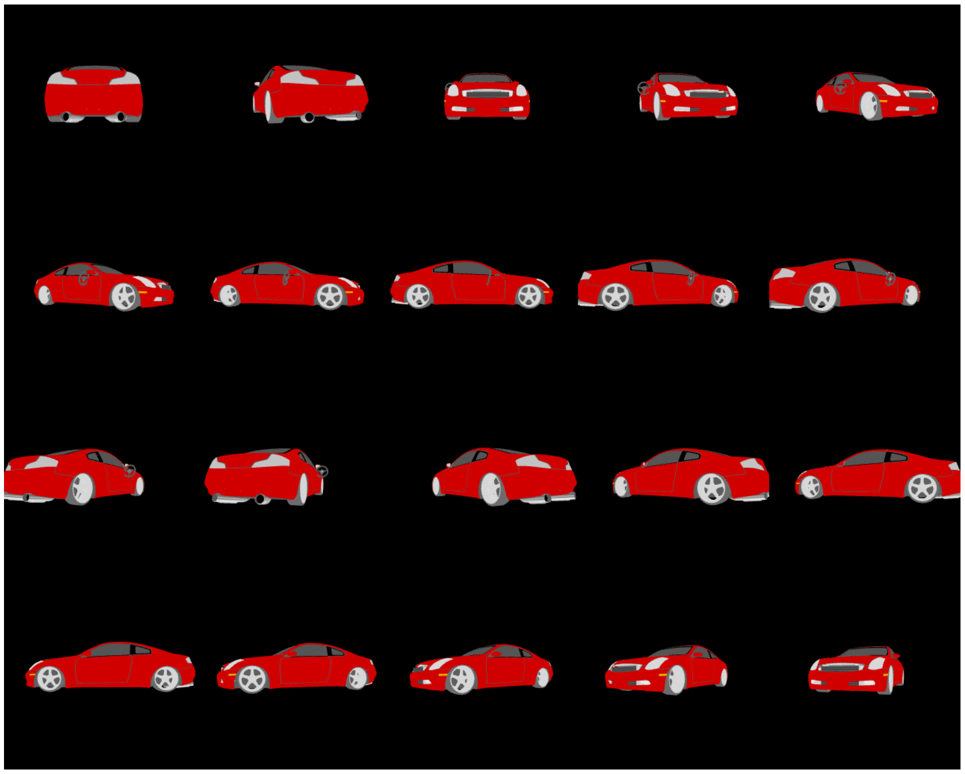

Figure C.1. A 5x4 grid of the same car from the 20 viewpoints used to generate our dataset..

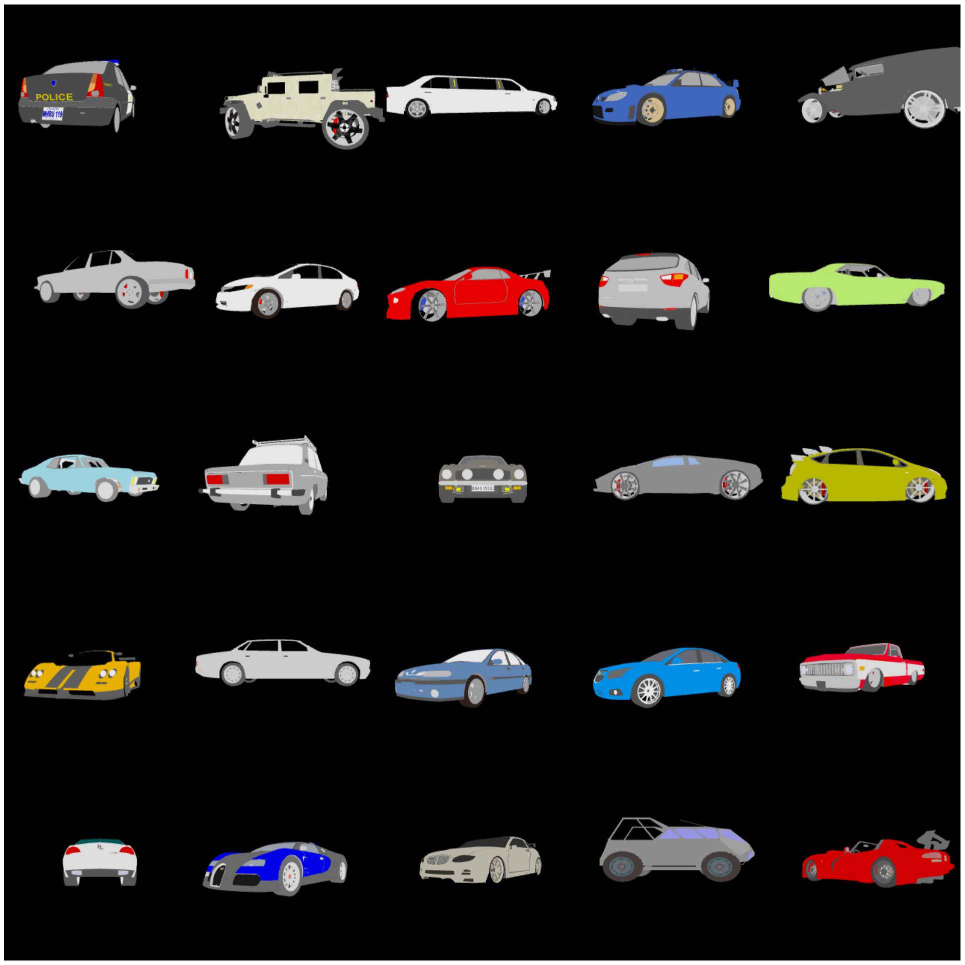

Figure C.2. A 5x5 grid of random samples from our synthetic cars dataset. Included to show the variety of cars represented..

APPENDIX D

Dataset - Planes

D.1 Introduction

As a final data set for our experiments, we synthesize another subset of the ShapeNet [10] dataset: planes (synset id: 02691156)

D.2 ShapeNet Planes

We synthesize each model in the shapenet planes synset from the same 20 views, rotated around its y axis in equal increments (see **??**). Additionally, to demonstrate the variety of cars represented in the dataset, see D.3.2.

Our dataset consists of 3535 3D models of planes, each rendered in the 20 viewpoints mentioned above for a total of 70700 samples. These are split into train, validation and test sets. We partition these samples into train, validation, and test sets. If a model is used in one of the sets, all 20 viewpoints are in that set.

Table D.1. Data Splits for Synthetic Shapenet Planes Dataset

Train	Val	Test
53960	8540	7480

D.3 Samples

D.3.1 Varying Rotation

D.3.2 Random Samples

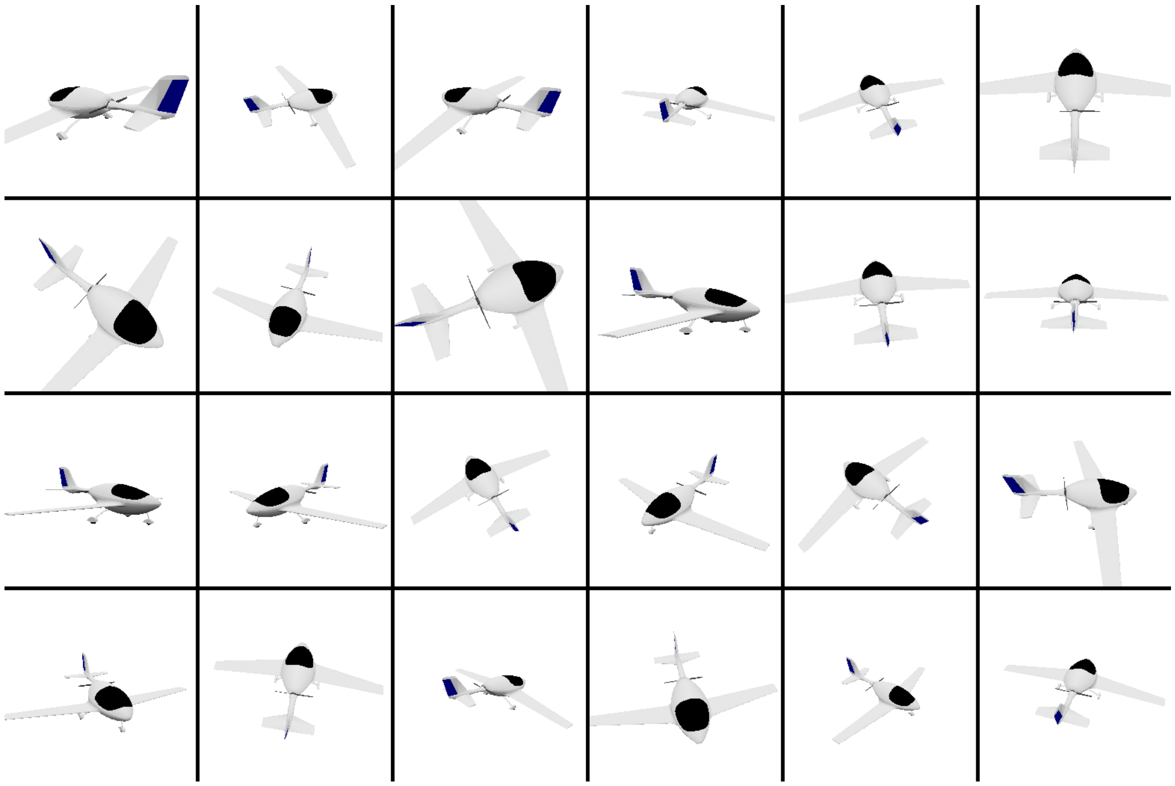

Figure D.1. A 5x4 grid of the same plane from the 20 viewpoints used to generate our dataset..

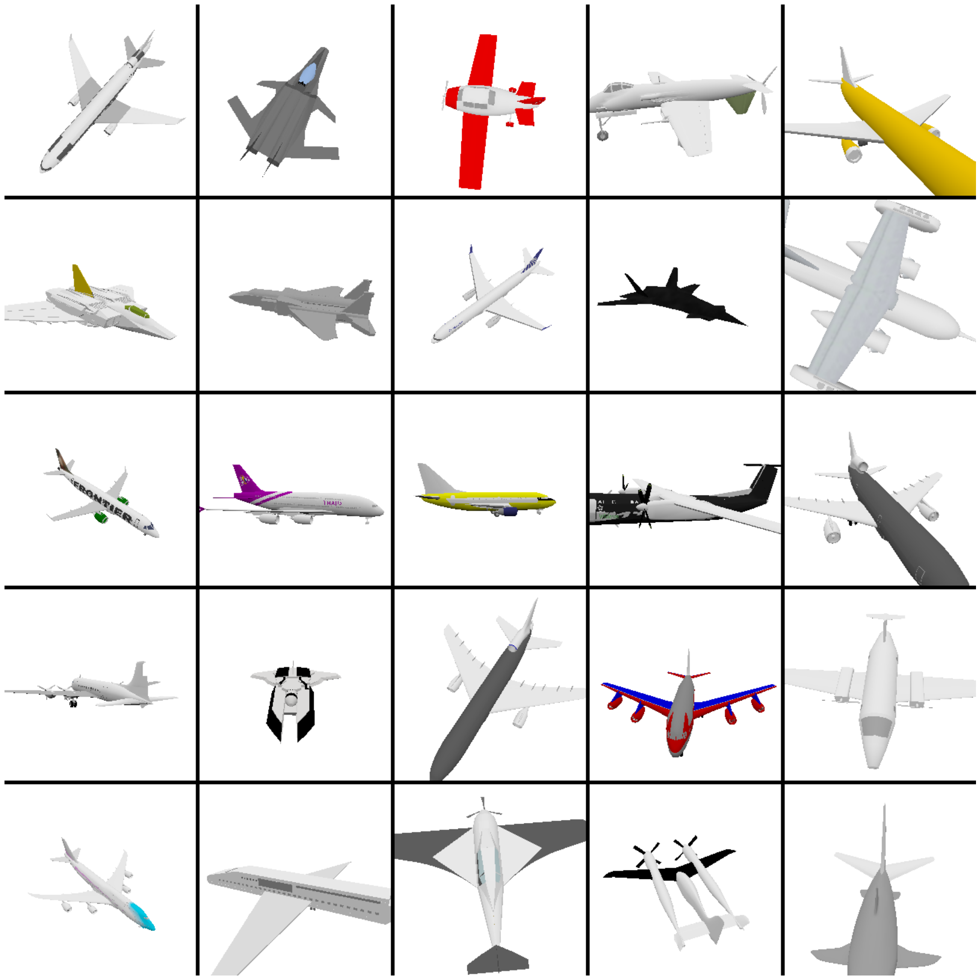

Figure D.2. A 5x5 grid of random samples from our synthetic planes dataset. Included to show the variety of planes represented..

REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[2] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[3] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *CoRR*, vol. abs/1511.06434, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:11758569

[4] N. Ravi, J. Reizenstein, D. Novotny, T. Gordon, W.-Y. Lo, J. Johnson, and G. Gkioxari, "Accelerating 3d deep learning with pytorch3d," *arXiv:2007.08501*, 2020.

[5] S. Liu, T. Li, W. Chen, and H. Li, "Soft rasterizer: A differentiable renderer for image-based 3d reasoning," *The IEEE International Conference on Computer Vision (ICCV)*, Oct 2019.

[6] R. Q. Charles, H. Su, M. Kaichun, and L. J. Guibas, "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.

[7] B. O. Community, *Blender - a 3D modelling and rendering package*, Blender Foundation, Stichting Blender Foundation, Amsterdam, 2022. [Online]. Available: http://www.blender.org

[8] P. Paysan, R. Knothe, B. Amberg, S. Romdhani, and T. Vetter, "A 3d face model for pose and illumination invariant face recognition," in *2009 Sixth IEEE International Conference on Advanced Video and Signal Based Surveillance*, 2009, pp. 296–301.

[9] Y. Bengio and Y. LeCun, Eds., *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. [Online]. Available: https://iclr.cc/archive/www/doku.php%3Fid=iclr2015:accepted-main.html

[10] A. X. Chang, T. A. Funkhouser, L. J. Guibas, P. Hanrahan, Q. Huang, Z. Li, S. Savarese, M. Savva, S. Song, H. Su, J. Xiao, L. Yi, and F. Yu, "Shapenet: An information-rich 3d model repository," *CoRR*, vol. abs/1512.03012, 2015. [Online]. Available: http://arxiv.org/abs/1512.03012

[11] W. Chen, J. Gao, H. Ling, E. Smith, J. Lehtinen, A. Jacobson, and S. Fidler, "Learning to predict 3d objects with an interpolation-based differentiable renderer," in *Advances In Neural Information Processing Systems*, 2019.

[12] H. Kato, Y. Ushiku, and T. Harada, "Neural 3d mesh renderer," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[13] C. Wah, S. Branson, P. Welinder, P. Perona, and S. Belongie, "Test," California Institute of Technology, Tech. Rep. CNS-TR-2011-001, 2011.

[14] W. Falcon and The PyTorch Lightning team, “PyTorch Lightning,” Mar. 2019. [Online]. Available: https://github.com/Lightning-AI/lightning

BIOGRAPHICAL STATEMENT

Jason Jennings was born in Dallas, Texas, in 1987. He received his B.S. degree from The University of Texas at Arlington in 2015, where he graduated summa cum laude. He continued his studies by directly entering the Ph.D. program at The University of Texas at Arlington to study Deep Learning and Computer Vision. He is the son of Sharon and Lawrence Jennings, brother to Justin Jennings, and the father of 4 rescue cats (Frank, Julius, Buddy and Sweetie).