BROWN
Computer Science

Transparent Voxelized Geometry Representations for Machine Learning

Henry Stone

Honors Computer Science Undergraduate Thesis

May 2020

Acknowledgements

First and foremost, I would like to thank my thesis advisor Professor James Tompkin for his guidance. He helped me understand the implications of my experiments and motivated me to pursue research in new and compelling directions. I would also like to thank my thesis reader, Professor Daniel Ritchie, for helping with this document and for his teaching in Graphics, which assisted with this work. Finally, I would like the thank my friends in the CS community for the way they have supported this project both explicitly through helping me generate ideas and implicitly through the camaraderie and support they have given me.

Contents

Chapter 1

Introduction

Neural rendering and neural object representations are becoming increasingly popular for view interpolation and other tasks. Many recent papers have created differentiable rendering frameworks, allowing one to transition from an object representation to a 2D, deep-feature image [1, 2, 3, 4]. In this work we explore the dynamics of a neural renderer for view interpolation.

The specific neural renderer we implemented is a two-part model. First, there is a learned representation of an object. Next, a rendering layer without any learned parameters converts this representation to a 2D feature image. Finally, the rendered features are up-scaled to produce a full-size image using a CNN. Using this model we sought to answer questions about the architecture of the network, its generalizability, and the structure of its learned representations.

Seeking an object representation and convolutional network that perform the best on view interpolation, we attempted to answer the following questions:

- Does a semi-2D object representation perform better than a fully 3D one?
- Does the structure of the output network significantly affect the models performance?

We also asked questions about the model's ability to generalize to new views and multiple scenes:

- Can a single output network be used with representations of different objects?
- Does augmenting the training data with affine transformations allow the trained model to be more robust?

Finally, we used our trained models to examine the properties of the feature space by comparing it to the feature space from DeepVoxels [1].

- Does an explicit opacity term allow the model to generalize to occlusions not seen in the training data?
- What is the relationship between the feature space and the object's surface appearance?
- Is the feature space rendered by the output network a legitimate target for learning?

Finally, we use the model we trained to create a real-time viewer for exploring rendered views. This demonstrates the usability of these models for real-time applications such as interactive editing or gameplay.

Chapter 2

Related Works

The field of rendering has long operated primarily on the forward process of taking an object representation and simulating light transport to calculate its appearance from a virtual camera. More recently, differentiable renderers have been developed [5, 6] which produce a backwards relationship between the object representation and the rendered view. This enables these renderers to optimize object representations in order to produce the desired view image. However, these renderers are limited by the fact that not all aspects of light transport are differentiable. For example, occlusion edges provide a discretized signal requiring various workarounds and relaxed assumptions to make all aspects of a renderer differentiable [7, 6].

Because of their reliance on gradients, neural networks require all of their components to be differentiable. Thus, as the graphics and computer vision communities have increasingly adopted neural networks, the differentiability of the rendering process has become ever more important.

One of the first works to do this was Visual Object Networks by [8]. Their network differentiably renders a 2.5D representation of a 3D occupancy grid. Their model first uses a 3D CNN to generate an object occupancy volume. This volume is then differentiably rendered to 2.5D at which point a 2D CNN is able to apply textures to the object. This separation of the object shape and texture allows the authors to control those two features independently. Additionally, by separating the operation, they can perform their spatial reasoning in a 3D framework and their textural reasoning in a 2D framework, matching the domain of the problem to the structure of the neural network used.

RenderNet by Nguyen-Phuoc et al. [4] also presents a differentiable renderer for an occupancy volume. Using this differentiable renderer, they learned to render the voxel grid smoothly, removing the hard edges caused by the binary occupancy. Additionally, they learned to simulate different classes of materials and rendering styles, such as the Phong lighting model or contour rendering.

Instead of just using a 3D volume to store occupancy information, DeepVoxels [1] stores object appearance properties in a voxel volume. The authors learned these features by integrating ground truth views of an object into their feature volume using an LSTM. While the image features are integrated into the volume, they trained a differentiable rendering network that learns to predict object occlusions and render the voxel features from different views of the object. Doing so allows them to perform high quality, spatially aware view interpolation on the rendered objects.

HoloGAN by Nguyen-Phuoc et al. [2] builds on the body of work which adds 3D reasoning to GANs. In addition to using a 3D neural network to produce an occupancy volume as per [8], they have their 3D network learn material features that are differentiably rendered to produce final images. Again, by separating the 3D reasoning and rendering, they can vary the view angles for a single object produced by the GAN, something which would be very difficult to do operating in 2D alone.

Neural volumes by Lombardi et al. [3] use their 3D representation to parameterize moving models in addition to just 3D shape. In an attempt to capture more detail in regions of high content, they use offset vectors to index into a second, warped voxel grid. Doing so allows their voxel grid to effectively have a

dynamic resolution, using more detail for a figure's facial features than for the surrounding space.

More recent works have moved towards more efficient representations of objects. Since voxels do not use most of their information to capture object surface details, Thies et al. [9] encodes neural features onto a model's texture. While this technique requires a proxy mesh, they show that a relatively low-quality mesh can still produce high-quality results.

Finally, NERF [10] and Scene Representation Networks [11] operate at the level of light rays. By using a neural network that maps from 3D space to local features, these models integrate along rays to produce their final appearance. The lack of explicit structure in the model features allows the network to encode details however it sees fit. This provides a powerful implicit regularization on the appearance of the rendered object, as the network has limited capacity to represent its different features and shapes. It is, in part, this regularization which allows the network to avoid overfitting to the training views. While these methods produce high-quality view interpolation results, they suffer from a lack of interpretability and editability, limiting their usefulness for other tasks. Additionally, these networks must run once for each 3d point sampled rather than once per ray, significantly increasing the cost of these algorithms.

Chapter 3

Dataset Creation

To train models to perform view interpolation, we need a dataset that provides ground truth images of objects with their corresponding camera poses. Since such data is difficult to capture accurately in the real-world, we opted to construct a synthetic dataset of object views.

3.1 Models and Scenes

This synthetic dataset is comprised of 12 models selected from the ShapeNet dataset [12]. These models were hand-selected in order to make sure they have correctly aligned normals and high-frequency detail. These two properties ensure that the rendered objects look reasonably realistic.

Each of these objects was used to construct a scene. Our volumetric rendering approach focuses on a cubic domain of interest extending from $(-0.5, -0.5, -0.5)$ to $(0.5, 0.5, 0.5)$. To ensure that each object fits within this domain, it is centered at $(0, 0, 0)$ and scaled to fit within the unit cube.

For simplicity, each object is lit by point lights. There are three bright point lights placed uniformly at random around the object to create sharp shadows. Additionally, ten weaker point lights are placed uniformly at random around the scene to simulate more diffuse lighting. Together, these lights provide a mix of lighting effects for our view interpolation networks to learn.

3.2 Camera Positions

For our 500 training viewing angles, we followed the work of Davis et al. [13] who captured unstructured light-fields using cameras positioned approximately on a sphere around each object. To do so, we selected uniformly sampled view angles and view distances varying between 2 and 5 units. To adequately capture each object, we pointed the cameras so that they were oriented towards $(0, 0, 0)$ and altered their FOV to capture the full unit cube.

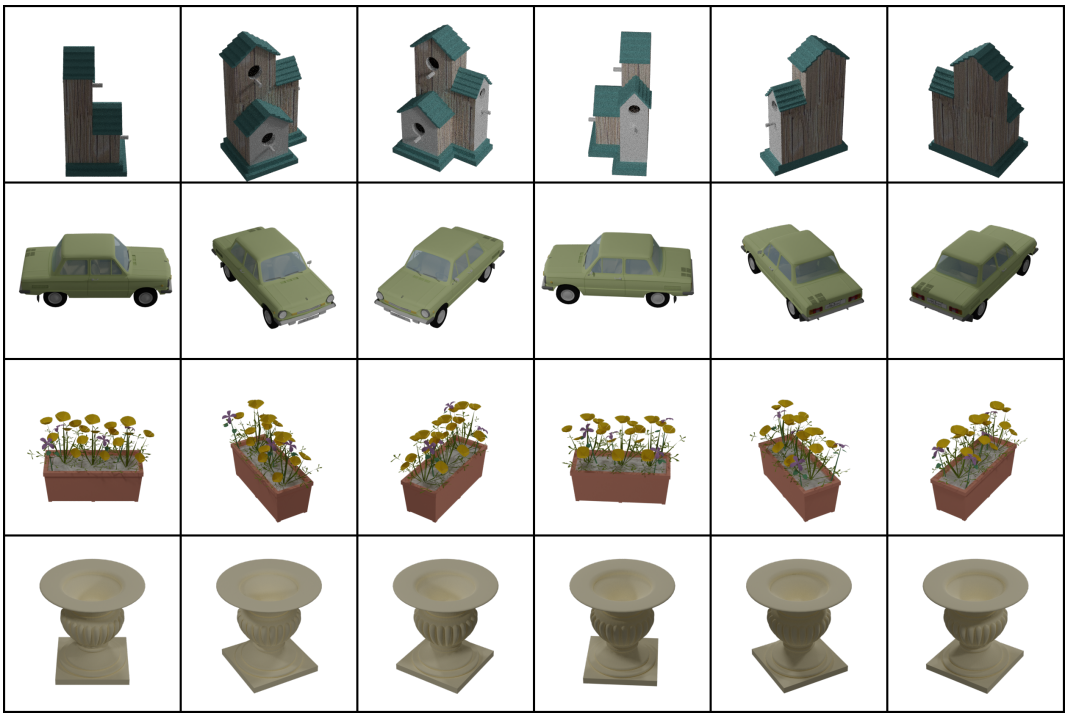
We rendered 100 testing viewing angles. For these images, we needed to be able to test for consistency across nearby views. In order to do so, we rendered an orbital set of views looking down at the object rather than the randomly sampled views used for training.

3.3 Rendering

We used the physically-based renderer Cycles, which comes packaged with Blender [14], to capture complex lighting phenomena in our renders.

For each object, we rendered its 600 views with 1000 samples per pixel. Each rendered image has a resolution of 512×512. Taken together, the views for each scene took approximately 8 hours to render on a GTX 1080 GPU.

(a) Example training images for each of the 12 scenes.



(b) Example testing images for 4 of the 12 rendered objects.

Figure 3.1: Example images from our view interpolation dataset

Chapter 4

Model Architecture Experiments

We chose a 2-part model to perform view-interpolation because of its simplicity. It separates the object representation, the neural renderer, and the network used to extract details without having additional components that complicate the analysis.

One of the main goals of neural rendering is to make the model architecture reflect the task it is solving. Thus, while the structure of a neural network is always significant, it is particularly important for these networks. To this end, this chapter examines how the architecture and training environment of a view interpolation model affect its performance.

First, we construct a canonical model against which to measure changes in the architecture. From there, we consider the dimensionality of the object representation and the architecture of our output network. Finally, we consider how the model performs when trained on different numbers of objects or with augmented input data.

4.1 Rendering Layer

While 3D structure is vital for operating on spacial properties in neural networks, real-world data is almost always captured in 2D or 2.5D. This divide presents an issue of how we can transition between the 2D and 3D representations of a scene. While, in general, there are many ways to do this, we seek a method that reflects the physical properties of a system and maximally takes advantage of our often limited 3D voxel resolution.

4.1.1 Rendering Framework

To make the physical properties of this rendering layer reflect real-world light transport, we separate the rendering of a neural volume into its features $\mathcal{F} \in \mathbb{R}^{f \times n \times n \times n}$ and its opacity $\mathcal{O} \in \mathbb{R}^{n \times n \times n}$. Here, n is the resolution of the voxelized volume being rendered, and f is the number of features being rendered. Additionally, we represent a pinhole camera using its camera-matrix, $\mathcal{C} \in \mathbb{R}^{3 \times 4}$. With this formulation, our rendering network takes the form of a function:

$$\mathcal{R}_{\text{volume}} : \quad \mathcal{F}, \mathcal{O}, \mathcal{C} \to \mathbb{R}^{f \times h \times w}$$

Here, (h, w) is the resolution of the pinhole-camera. In the real-world, we expect there to be consistent rendering rules across all pixels in an image. Thus, we can simplify the problem to a per-pixel rendering using the starting location, $\mathcal{S}_{ij} \in \mathbb{R}^3$, and direction, $\mathcal{D}_{ij} \in \mathbb{S}^2$, of each pixel's central ray. This reparameterization allows us to consider the simpler problem of rendering an individual ray:

$$\mathcal{R}_{\text{ray}} : \quad \mathcal{F}, \mathcal{O}, \mathcal{S}_{ij}, \mathcal{D}_{ij} \to \mathbb{R}^f$$

From this representation, we need to find a weighted sampling of $\mathcal{F}$ based on our ray. One common approach is to sample points equidistant along the ray defined by $(\mathcal{S}_{ij}, \mathcal{D}_{ij})$ [1]. While this technique allows us to use well-studied point sampling techniques, it requires a dense enough set of points to capture the approximate amount of time the ray spends in each voxel. Sampling this many points can impose a serious runtime cost and limit the frame rate of the neural rendering. Instead, we analytically compute the ray/voxel intersections, resulting in a set of the voxels each ray intersects represented by their indices: $\{V_{nij} \in \mathbb{Z}^3\}$ and the distance the ray travels within each intersected voxel: $\{D_{nij} \in \mathbb{R}^+\}$ (Figure 4.1). This representation uses a minimum number of values to fully represent the voxels encountered by a ray at maximal resolution. Using the voxels encountered, we can index into the tensors representing our volume, giving us features, $\{\mathcal{F}_{nij} \in \mathbb{R}^f\}$, and opacities, $\{\mathcal{O}_{nij} \in \mathbb{R}\}$.

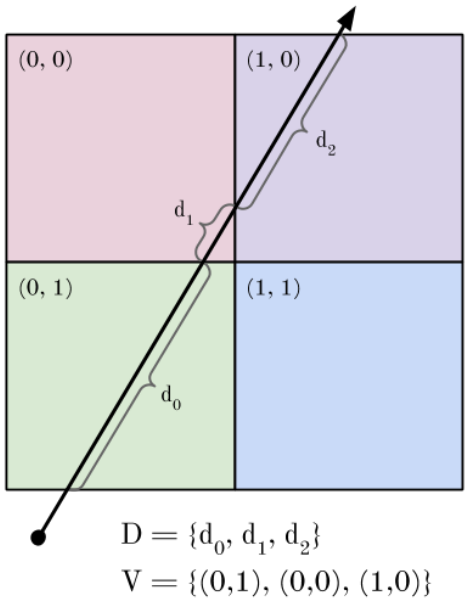


Figure 4.1: Ray-volume intersection collects which voxels are intersected and the distance the ray remains in each voxel.

After sampling a set of features, DeepVoxels [1] uses a neural network to weight these samples and predict the final value for the ray. In our rendering network, we use our explicit opacity and consider the amount of light passing through a voxel to exponentially decay with distance in the volume per the Beer-Lambert Law.

First, we scale $\{\mathcal{O}_{nij} \in \mathbb{R}\}$ to ensure that it lies in $(0, 1)$ using a sigmoid activation. Next, we consider this to be the fraction of light passing through a 1-unit distance in the medium. Using this, we compute the amount of light that passes through the given distance in the medium as follows:

$$\text{Transmission}_{nij} = \exp\left[D_{nij} \cdot \log(S(\mathcal{O}_{nij}))\right]$$

The visibility of each voxel is then the product of how much light that voxel blocks and the fraction of light absorbed as it passes through other voxels to the camera center:

$$\text{Vis}_{nij} = \left(\prod_{k=o}^{n-1} \text{Transmission}_{kij}\right)(1 - \text{Transmission}_{nij})$$

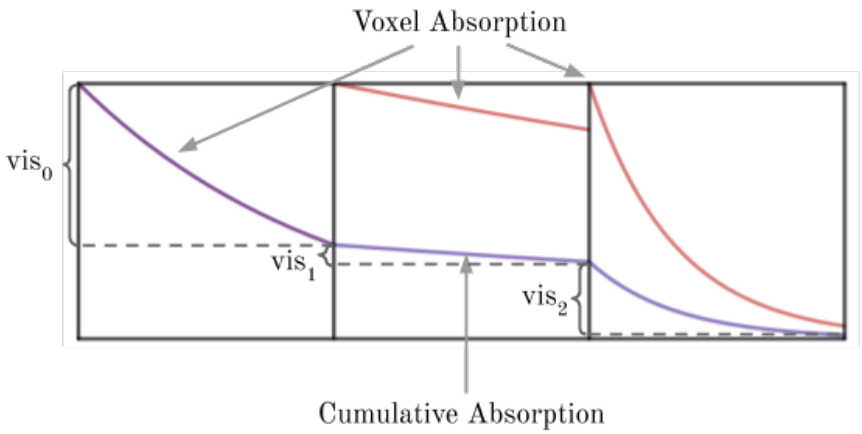


Figure 4.2: For a ray coming from the left of the figure, visibility of each voxel is calculated based on the cumulative absorption of the voxels.

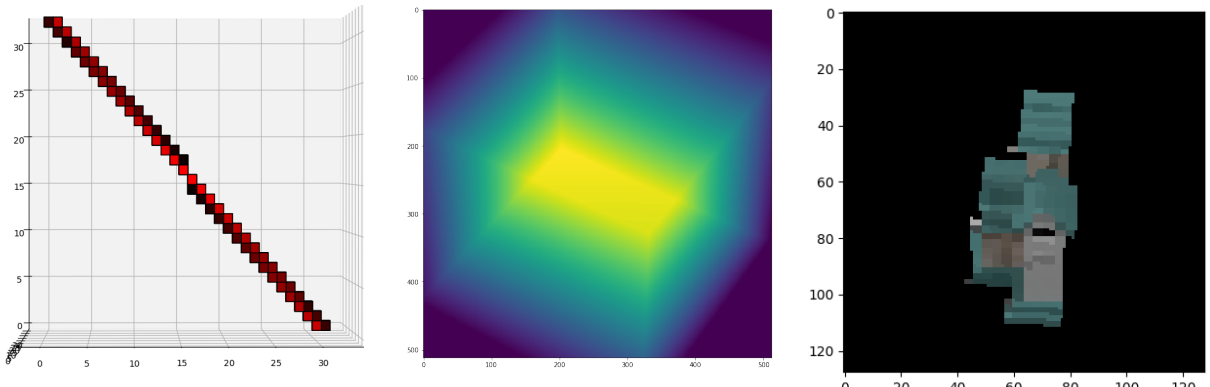
Using these visibilities we can take a principled weighted average of the features in each voxel:

$$\mathcal{R}_{\text{ray}} = \sum_{n} \text{Vis}_{nij} \cdot \mathcal{F}_{nij}$$

Because we are treating all objects as semitransparent, we have no hard occlusion boundaries, which would result in non-differentiable rendering. Thus, the rendering network enables us to differentiably convert between a 3D object representation and a 2D view of that representation as taken by a pinhole camera.

4.1.2 Implementation

Since we are operating on fixed views, we do not need the rendering operation to be differentiable with regard to the camera matrix. We need only ensure that derivatives are propagated to the feature and opacity volumes. Doing so allows us to use precomputed values for $\{V_{nij}\}$ and $\{D_{nij} \in \mathbb{R}^+\}$, which we generated in a Python script.



(a) Voxels intersected by a specific ray. Color indicates the time spent in each voxel.

(b) Total time each ray spends intersected with the unit cube volume.

(c) Rendering of voxelized treehouse using binary visibility term.

Figure 4.3: Visualized rendering layer intersections.

While this was sufficient to show that the model worked, the high cost (several minutes) to compute each

camera voxel intersection set severely limited the usability of the rendering layer to produce novel views. This challenge led to a C++ and a CUDA implementation of the camera / voxel-grid intersection code. This speedup allowed the intersections to happen during training or inference rather than being precomputed, with the rendering running at 150+ FPS for a 64×64 image of a $32 \times 32 \times 32$ voxel grid.

Once we have the intersection indices and distances, we use PyTorch [15] to differentiably calculate the rendered features by using the functions outlined in the section above. Since we do not need to propagate derivatives to the camera matrices, we did not produce a differentiable kernel for these intersections, although, in principle, one could do so.

4.2 Losses

Many papers focused on neural rendering use adversarial losses in order to produce high-quality renderings of objects [1, 2]. While models trained this way undoubtedly produce higher fidelity images, the adversarial loss has several issues. First, since the discriminator networks are trained only on individual images, they do not guarantee view consistency in high-frequency content. We observe this high-frequency content to stand relatively still while the larger model moves underneath it.

Additionally, adversarial losses can obscure how effectively each network is capable of propagating information from the object representation to the final image. Thus, we opted for a simple L2 loss. For the transparency of the image, we use standard MSE. However, for the loss on the RGB channels of the images, we do not care what colors the network predicts for transparent image regions. Thus the loss is weighted by the opacity of the ground truth image.

The ability to use transparency loss is an artifact of our synthetic dataset and is not an option for real-world scenes. Provided for comparison at the end of this chapter in figure 4.13 are loss values for a model that only produces RGB images. For this model, loss is computed only over the RGB portions of these images using an unweighted MSE loss.

4.3 Canonical Voxel-Volume Based Model

In order to investigate what factors are important in creating a voxelized representation for machine learning, we created a primary neural network from which we could test modifications. This model, hereafter referred to as the canonical model, is trained to do view interpolation.

In the canonical model, there are three sets of parameters trained using the Adam optimizer[16].

1. A voxelized feature volume with resolution $32 \times 32 \times 32$ and 64 features per voxel.
2. A voxelized opacity volume with resolution $32 \times 32 \times 32$ and a single opacity value per voxel.
3. An output U-Net [17] which upscales from a resolution of 64×64 with 64 features to a 4-layer transparent image with resolution 512×512.

The input to this network is a camera matrix $\mathbf{C} \in \mathbb{R}^{3\times4}$. Using this camera matrix, we generate a set of 64×64 rays which feed into the rendering network (Section 4.1.1), producing a rendered image of the feature and transparency volumes with resolution 64×64 and 64 features per pixel.

This feature image is then upscaled from 64×64 to 512×512 by an imbalanced U-Net architecture (the output network). Finally, the network is trained using an L2-loss between the predicted view and the ground truth view. The RGB portion of the loss is weighted by the ground truth transparency in order to encourage consistency only over the object itself (See section 4.2).

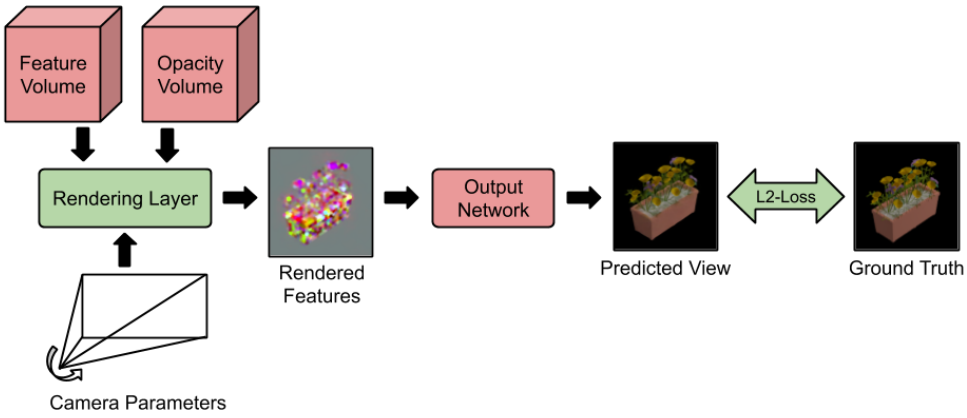


Figure 4.4: Network architecture for canonical volume based model.

In order to demonstrate the capability of the canonical model, we trained it on the treehouse and green car objects. In addition to showing the successes and failures of the transparency based volume interpolation, it serves as a baseline for comparison with other models and understanding deep voxel feature spaces.

Training for the canonical took 33 hours on a GTX 1080, representing 3000 total epochs of training.

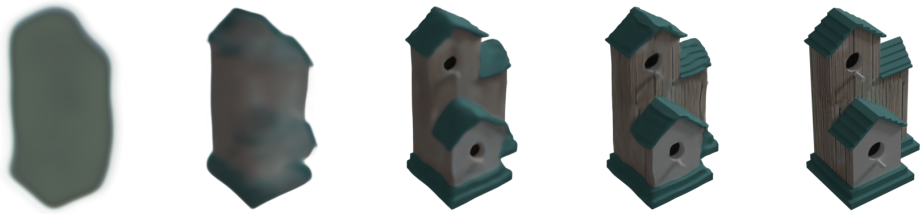


Figure 4.5: The progression of testing images at different points during training (10, 100, 500, and 2400 epochs). Ground truth at the right.

Here we see that the network first learns the shape of the object, and once it has learned the shape, it is then able to propagate the model color back through the network to discern coarse features in the volume. Finally, the high-frequency features are learned within the U-Net in order to upsample the resolution of the volume to produce final images.

Figure 4.6: Testing images for several scenes compared to ground truth.

While from a distance the rendered views appear to be high quality, overfitting to the training set results in warped images as the U-Net is unable to represent all surface patches from the exact desired angles.

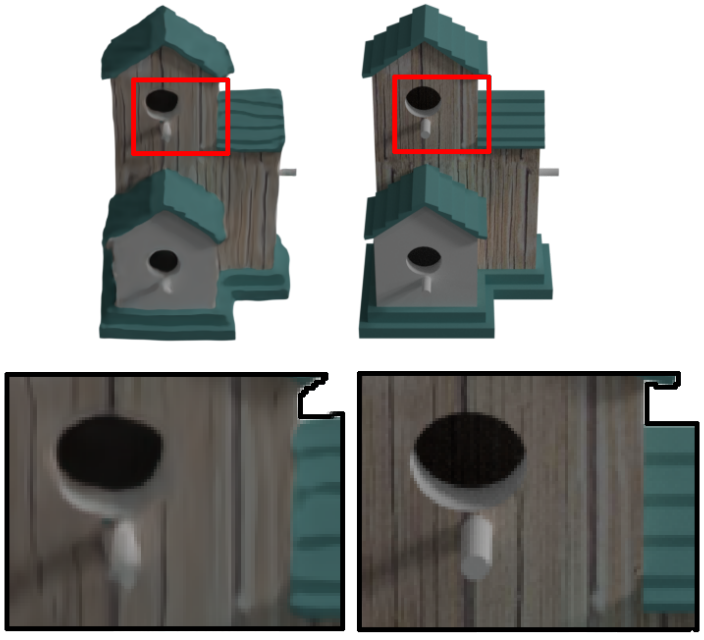


Figure 4.7: Close up comparison of ground truth and rendered testing view.

Here we see that high frequency details are misplaced slightly. This is likely due to overfitting and the low resolution $32 \times 32 \times 32$ of the opacity voxels.

4.4 SRGAN Output Network

Question: Does the structure of the output network significantly affect the models performance?

Specific model architectures significantly impact many machine learning problems. It is thus natural to ask whether the architecture of the output network profoundly impacts this view interpolation model. We want an output network that is capable of producing high-frequency image patches from a lower resolution feature image. A natural choice for neural network architecture comes from super-resolution networks because of their ability to both upscale and produce a broad set of high-frequency details.

For our specific implementation, we used the network architecture from "Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network" [18], frequently referred to as SRGAN. To adapt this network to our upsampling problem, we modified the first layer to take in a feature image with 64 features, consistent with what is output by the rendering layer in the canonical model. Otherwise, the output network is identical to a 3-block SRGAN generator network designed to upscale images by a factor of 8.

While the errors from this network were undoubtedly of a different character than those with the U-Net output network, the actual losses for the SRGAN model are very comparable to those from the canonical model. (Note: because SRGAN produces RGB images we compared to an RGB-only version of the canonical model).

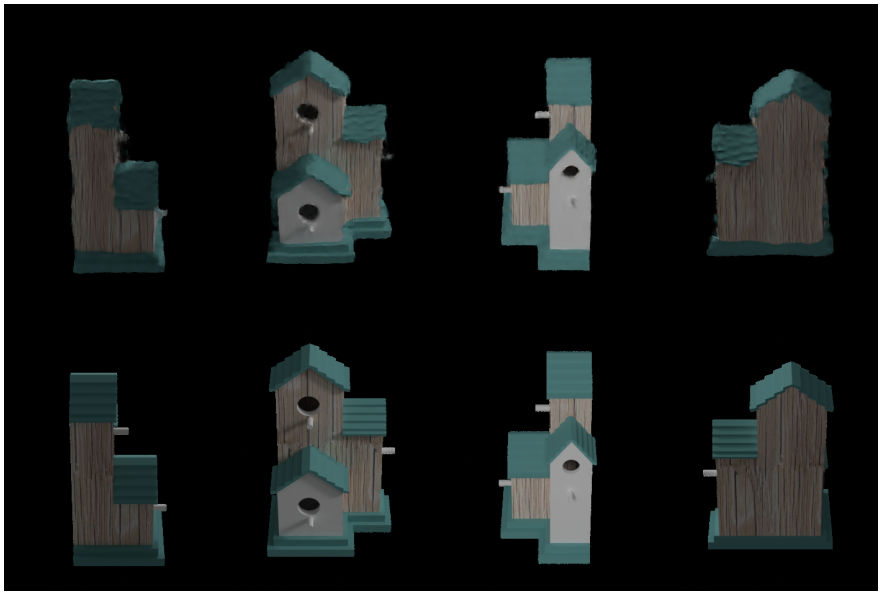


Figure 4.8: Testing results for SRGAN output network (top) compared to ground truth (bottom).

Model Type	Training		Testing	
	RGB Loss	Transparency Loss	RGB Loss	Transparency Loss
Canonical	3.83×10^{-5}	2.81×10^{-5}	23.15×10^{-5}	254.50×10^{-5}
RGB Only Canonical	4.56×10^{-5}		65.80×10^{-5}	
SRGAN	3.37×10^{-5}		68.15×10^{-5}	

These results suggest that perhaps the architecture of the output network is not particularly significant in determining the model's performance on view interpolation.

4.5 Texture vs Volume

Question: Does a semi-2D object representation perform better than a fully 3D one?

One possible way to improve the quality of view interpolation results is to pass better, higher-frequency feature images to the output network. In the volume-based model, we are limited to a $32 \times 32 \times 32$ feature space because memory use grows cubicly with resolution. Most of this memory goes effectively unused, representing the space around the object. We see this limitation in Neural Volumes [3], where they use various techniques to increase the effective resolution of their voxel space, improving the quality of their renderings.

The fundamental issue is that a dense 3D representation is used for an object whose surface texture is only two dimensional. One way that we can strive to improve the resolution of our object representation is to use 2D textures instead of a 3D feature volume.

We represent each object using three axis-aligned images, $\{I_{xy}, I_{xz}, I_{yz}\}$. Each of these images is 128×128, allowing much higher resolution than the feature volume with approximately the same total number of features. To sample the features at a given location in 3D space, we concatenate the sampled features as seen in figure 4.9

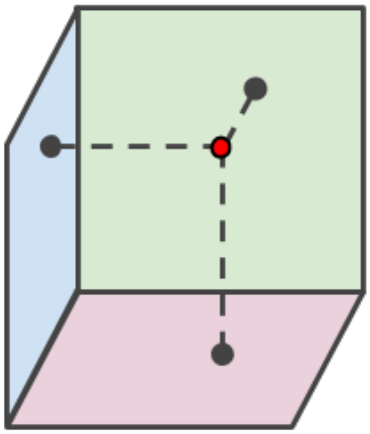


Figure 4.9: The three feature textures are sampled by projecting the 3D point orthogonally to each plane and bilinearly sampling those points on the textures.

In order to decide where to sample these textures, we train a 3D opacity volume— performing ray voxel intersections using our CUDA kernel from section 4.1.1. For each voxel, we take the midpoint of the ray within that voxel. We then take a weighted average of these voxel midpoints based on each voxel's cumulative visibility from the camera. This method allows us to find the subvoxel ray/object hit locations. It is, however, still limited by the $32 \times 32 \times 32$ resolution of the opacity volume.

At training time, we see that this additional resolution allows the network to capture the object shape and details with as much accuracy as the 3D networks while running in approximately half the time. However, having additional weights to represent the surface features of the object appears to be a double-edged sword as it results in stronger overfitting and worse testing results.

Model Type	Training		Testing	
	RGB Loss	Transparency Loss	RGB Loss	Transparency Loss
Canonical	3.83×10^{-5}	2.81×10^{-5}	23.15×10^{-5}	254.50×10^{-5}
Texture Model	3.46×10^{-5}	1.53×10^{-5}	106.45×10^{-5}	1260.00×10^{-5}

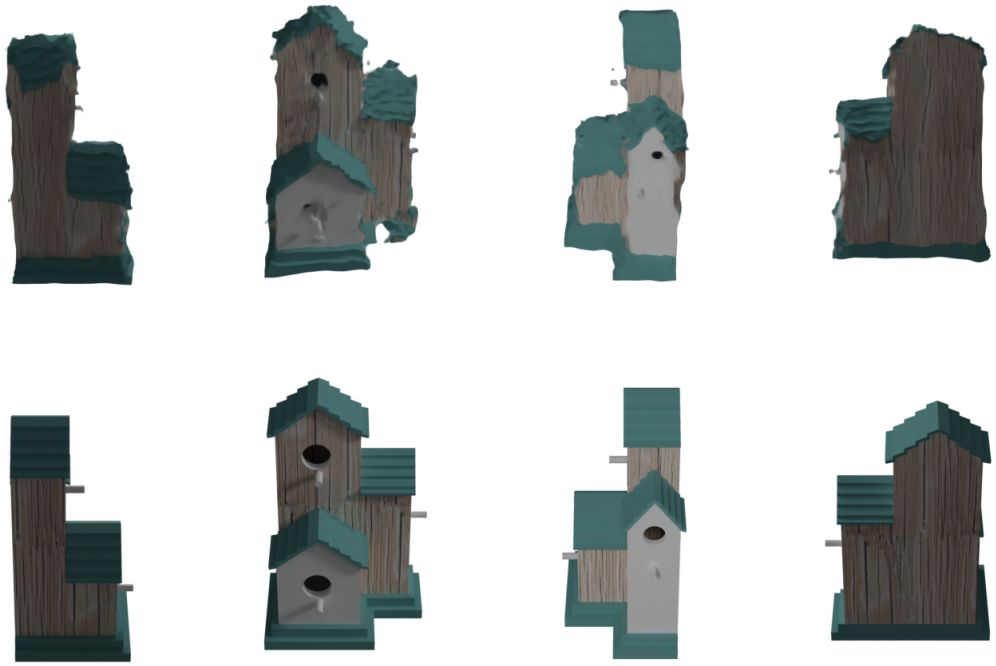


Figure 4.10: Testing results for the texture based network show massive distortion around the edges of the model.

4.6 Multi-Object Rendering

Question: Can a single output network be used with representations of different objects?

There are several reasons we might want to train a view interpolation network on multiple object volumes with a single output network. First, the output network generalizing to multiple objects may reduce overfitting for individual objects. Second, if we can train an output network that is sufficiently generalizable, we can hold it constant while training the feature and opacity volume of a new object. Training of this kind is discussed in section 5.3

The modification to the network is relatively simple. Rather than having a single feature and opacity volume, we train several, choosing which one to render based on the scene the training view was drawn from. Training this model, however, we observe that the network loses the ability to represent fine details when trying to render more than one object.

	Training		Testing	
# Objects	RGB Loss	Transparency Loss	RGB Loss	Transparency Loss
1	3.83×10^{-5}	2.81×10^{-5}	23.15×10^{-5}	254.50×10^{-5}
2	6.37×10^{-5}	3.92×10^{-5}	24.20×10^{-5}	249.00×10^{-5}
4	15.70×10^{-5}	46.00×10^{-5}	35.80×10^{-5}	428.00×10^{-5}

Here we see that the training loss is substantially higher for the network trained to represent two objects,

and even more so for the model representing four objects. Since each of the objects has its own feature and opacity volume, this bottleneck must be a result of the output network, not the object representation. It seems that despite having separate feature volumes for each object, the U-Net cannot represent the necessary mapping to high-resolution image patches.

We see the same pattern of increase, although to a lesser extent, when looking at the testing losses. Perhaps there is some merit to the theory that training on multiple models reduces overfitting; however, the limitations of the multi-object model vastly overwhelm any improvements.

We can see the reduction in visual quality for more objects in figure 4.11 especially when compared to the results of the canonical model in figure 4.6.

(a) Testing views of model trained on 2 objects.

(b) Testing views of model trained on 4 objects.

Figure 4.11: Comparison of testing results for models trained 2 and 4 objects show that quality goes down the more objects are being represented.

4.7 Data Augmentation

Question: Does augmenting the training data with affine transformations allow the trained model to be more robust?

In general, we expect models of the type shown in Section 4.3 to be able to represent all of the views similar to those on which they were trained. However, for view interpolation to be useful, sometimes we wish to consider camera parameters significantly different from those used during training. Unfortunately, our loss does not enforce that the network can represent these camera poses. In some cases, such as views revealing otherwise occluded content, our training data will be insufficient to determine the appearance of the object. However, in cases of the camera having a different FOV or rotation, an affine image space transformation of a training image could have predicted how the object would appear.

In these cases, we must take additional steps in order to ensure that the output network can represent these object surface patches. To do this, we augment the training data to include image space affine transformations for training views. A bonus of this type of augmentation is that it reduces overfitting to the training set, as the output network needs to learn a more robust mapping from feature space to image space.

In general, we consider an affine transformation in homogeneous coordinates mapping from the training view to a new view of the form:

$$\mathbf{A} = \left[\begin{array}{c|c} A_{2\times 2} & \vec{t} \\ \hline \vec{0} & 1 \end{array}\right] \in \mathbb{R}^{3\times 3}$$

To produce the new training view we modify our input camera matrix as follows $\mathbf{C}' = \mathbf{AC}$. We also modify the ground truth image by resampling our original training image using the above affine transformation and bicubic interpolation.

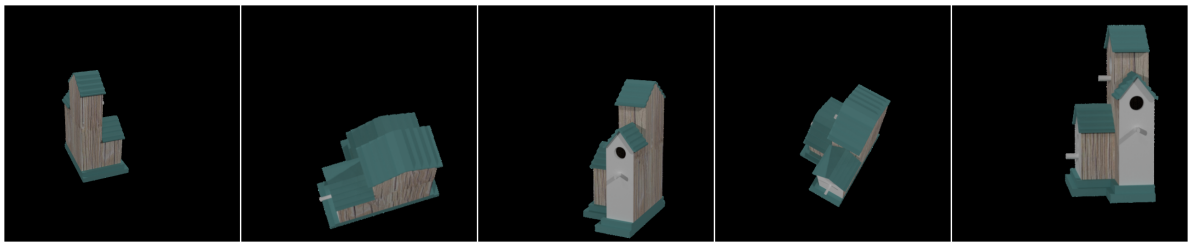


Figure 4.12: Example affine warped ground truth training images with translation and scale.

Adding these additional training views teaches the U-Net to represent object patches at scales and rotations not present in the original training data. This assists in extrapolating new views and avoiding overfitting during training. In general, this provides robust results that are used later for volume editing in section 5.2.

Model Type	Training		Testing	
	RGB Loss	Transparency Loss	RGB Loss	Transparency Loss
Canonical	3.83×10^{-5}	2.81×10^{-5}	23.15×10^{-5}	254.50×10^{-5}
Affine Transformations	7.17×10^{-5}	11.95×10^{-5}	7.22×10^{-5}	38.25×10^{-5}

Here we see the drastically reduced overfitting using the model augmented with affine transformations. With the reduction in testing loss, the model trained with affine transformations performs best out of all of our models during testing (Figure 4.13).

We note that this augmentation is necessary because the output network is capable of operating on image patches rather than individual rays. For a network that predicts an object's surface appearance based solely on an individual ray, this augmentation would result in the same samples being passed to the network multiple times and would thus be unnecessary. We see this in the Neural Radiance Field [10] network, which has a significantly reduced opportunity to overfit to specific surface patches because the network operates on individual pixels.

Model Type	Training		Testing	
	RGB Loss	Transparency Loss	RGB Loss	Transparency Loss
Canonical	3.83×10^{-5}	2.81×10^{-5}	23.15×10^{-5}	254.50×10^{-5}
2 Objects	6.37×10^{-5}	3.92×10^{-5}	24.20×10^{-5}	249.00×10^{-5}
4 Objects	15.70×10^{-5}	46.00×10^{-5}	35.80×10^{-5}	428.00×10^{-5}
RGB Only Canonical	4.56×10^{-5}		65.80×10^{-5}	
SRGAN	$\mathbf{3.37 \times 10^{-5}}$		68.15×10^{-5}	
Texture Model	3.46×10^{-5}	$\mathbf{1.53 \times 10^{-5}}$	106.45×10^{-5}	1260.00×10^{-5}
Affine Transformations	7.17×10^{-5}	11.95×10^{-5}	$\mathbf{7.22 \times 10^{-5}}$	$\mathbf{38.25 \times 10^{-5}}$

Figure 4.13: Comparison of losses for all tested models

Chapter 5

Representation Analysis

The applications of neural rendering extend far beyond view interpolation. Ideally, we would also like to use neural object representations for segmentation, editing, and classification. These applications are all dependent on the structure of the network's learned representations.

For example, in order to be able to edit a feature volume, we expect that certain features correspond to specific visual properties. If this were not the case, a user who copies a part of the volume somewhere else might observe an unexpected result.

5.1 Volume Features vs DeepVoxel Features

Question: What is the relationship between the feature space and the object's surface appearance?

An interesting point of comparison for the voxel spaces we generate is the voxel structure arising from DeepVoxels [1]. DeepVoxels have the property that their feature volume is the result of images being projected through and accumulated by a recurrent neural network. Because the feature volume is a direct function of the input images, we expect it to correspond to features in image space. However, because the recurrent neural network is trained on each object individually, we expect the training to somewhat shape the features into whatever configuration is most optimal for rendering correct results. This may or may not cause volume features to correspond to image features.

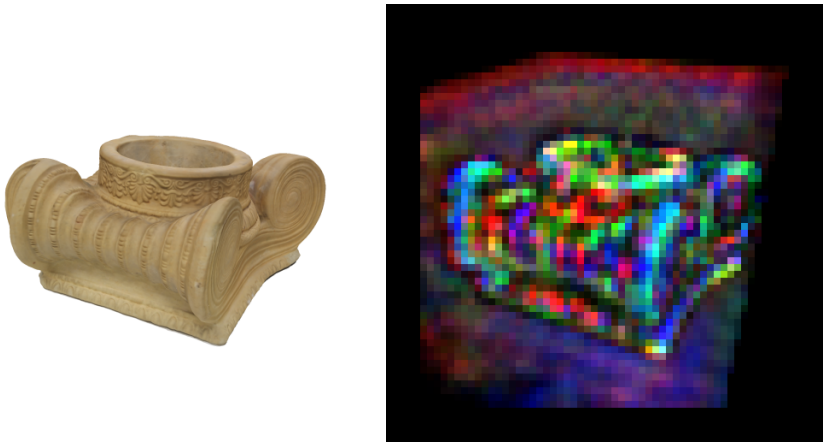


Figure 5.1: Predicted view using DeepVoxel architecture alongside a rendering of the feature space (Projected to 3 dimensions using PCA)

In figure 5.1, we see that DeepVoxels have the expected property that there are consistent features present across similar regions of the object. For example, the round edges of the Greek column are all blue, indicating that blue is used consistently to represent curved edges. By contrast, the voxel features of our network (figure 5.2) are solely a result of gradient descent and tend to be less consistent across similar image features.

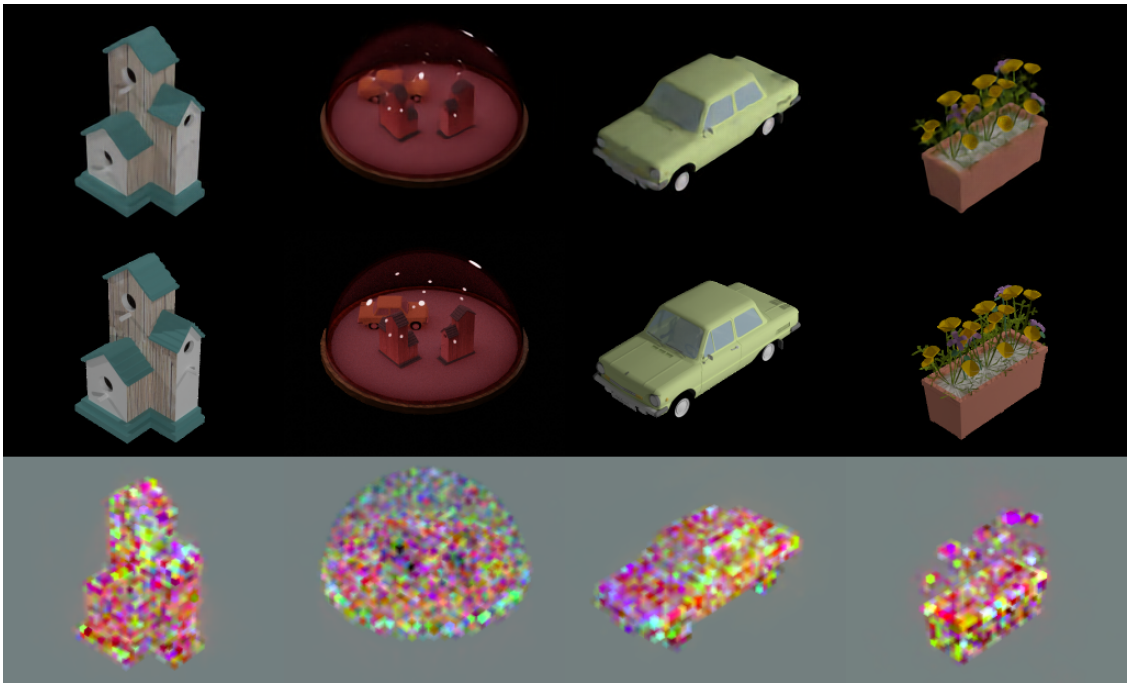


Figure 5.2: Predicted object views (top), ground truth object views (middle), rendering of voxel space using the rendering layer projected to 3 dimensions using PCA (bottom).

We hypothesize that when features are not required to be a function of input images, they are just used to distinguish different patches on the object surface. Once the local patches are identifiable, the output network learns a mapping from these individual patches to image patches in the predicted view. While this mapping is valid for reproducing a single object, it would scale very poorly as the number of objects using a single output network increases (as in section 4.6). This process is observed in our multi-object model,

which performed worse with larger numbers of objects.

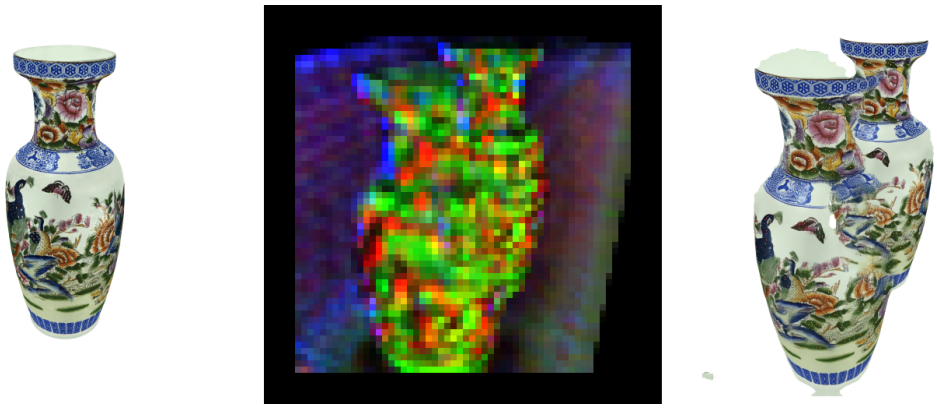
5.2 Volume Space Edits

Question: Does an explicit opacity term allow the model to generalize to occlusions not seen in the training data?

As mentioned above, one common thing we might want to do with neural volumes is making edits. There are many facets of what makes edits work as expected, but one significant factor is the ability to handle novel occlusions.

As always, with a neural renderer trained on a single object, we expect any deviations from the object to be handled at least somewhat poorly. However, we expected that having an explicit transparency term within our rendering equation would outperform the trained neural network occlusion model used in [1].

In order to test this, we replicated the features for an object side by side within the feature volume. We then passed this new duplicated set of features through the output network to produce a test view for the edit.



(a) Test rendering of duplicated DeepVoxels vase features shows occlusion being improperly calculated.

(b) Rendering of duplicated treehouse model using explicit transparency volume with ground truth treehouse on the right. While errors are present, no large scale occlusion errors occur.

Figure 5.3: Comparison of occlusion in duplicated models for DeepVoxels and the our model with explicit transparency.

Another benefit of an explicit opacity volume is that it gives us a mechanism for merging two overlapping objects. Specifically, we can take features from each voxel weighted proportionally to their transparency. Doing so allows us to capture volume intersections as in figure 5.4.

Figure 5.4: Overlapping edit of two treehouses (left) compared to ground truth single treehouse (right).

While explicit transparency does allow basic edits like this, we are still restricted to image patches that the output network learned to represent during training. We see this limitation very clearly when we try to produce an edit with much more complex occlusion.

(a) Treehouse model testing view. (b) Snowglobe model testing view. (c) Combined model.

Figure 5.5: The output network cannot reproduce image patches of the treehouse inside the snowglobe because it did not observe them during training.

To be capable of performing edits like the one shown above, the output network would need to represent a generalized mapping from features space to output patches rather than one which works only for the small number of observed objects.

5.3 Training Into Feature Space

Question: Is the feature space rendered by the output network a legitimate target for learning?

In section 4.6, we saw that as the number of objects increases, the output network loses the ability to produce fine details of objects. Given this, we expect that these networks would not be able to represent new objects well without having been trained on them.

Nonetheless, to see how generalizable these multi-object networks are, we decided to train a feature volume to perform view interpolation on a new scene while holding the weights of the output network from the multi-object model constant.

We observed two effects. First, the testing quality of images produced without varying the output network was much lower compared to images from the canonical model or the multi-object network. This reduction in quality indicates that the output network is indeed overfitting, not just to view angles, but also to the scenes observed during training.

Figure 5.6: Comparison of model view (left) produced without altering the output network compared to ground truth (right).

The second observation is that while in general, these models take many fewer iterations to converge (5000 instead of 150000), how long they take to converge is dependent on how long the original multi-object model was trained. For multi-object models trained to completion, it took many more iterations to produce a result of any quality at all. This suggests that the further into training the multi-object model is, the more it overfits to only the objects for which it was trained. The overfitting makes the feature space less friendly to gradient descent optimization, as it is significantly discontinuous.

Despite the low quality of the above image, the low training time suggests that a more generalizable output network would allow a very effective way to train neural voxel volumes.

Chapter 6

Real Time Viewer

Our model benefits from three features, which make it particularly fast relative to other neural rendering models. First, the ray marching step sizes, and feature values are specified explicitly by out rendering step. Thus, there is no costly neural network component for predicting these at each sampled location [11, 10]. Second, rather than densely sampling space to capture our volume, we explicitly compute the time spent within each voxel, reducing the number of samples needed to represent a ray's path [1, 10]. Finally, we sample our voxel volume at a lower resolution than the final image allowing a CNN to predict the final image appearance with fewer voxel ray intersections. In combination, these speedups allow our model to run in real-time.

In order to show the benefits of real-time view prediction relative to offline view interpolation, we created a real-time viewer. On an RTX 2080ti, we were able to render novel 512×512 views of objects at approximately 80 frames per second. On a much weaker MX 150 GPU, we were able to render views at approximately 15 frames per second.

The viewer allows the user to move the object around in real-time, viewing it from many different angles. The viewer helps explore any systematic errors present in the neural renderer. It also shows how neural rendering could plausibly be used in many real-time applications, including model editing, video games, and movie creation.

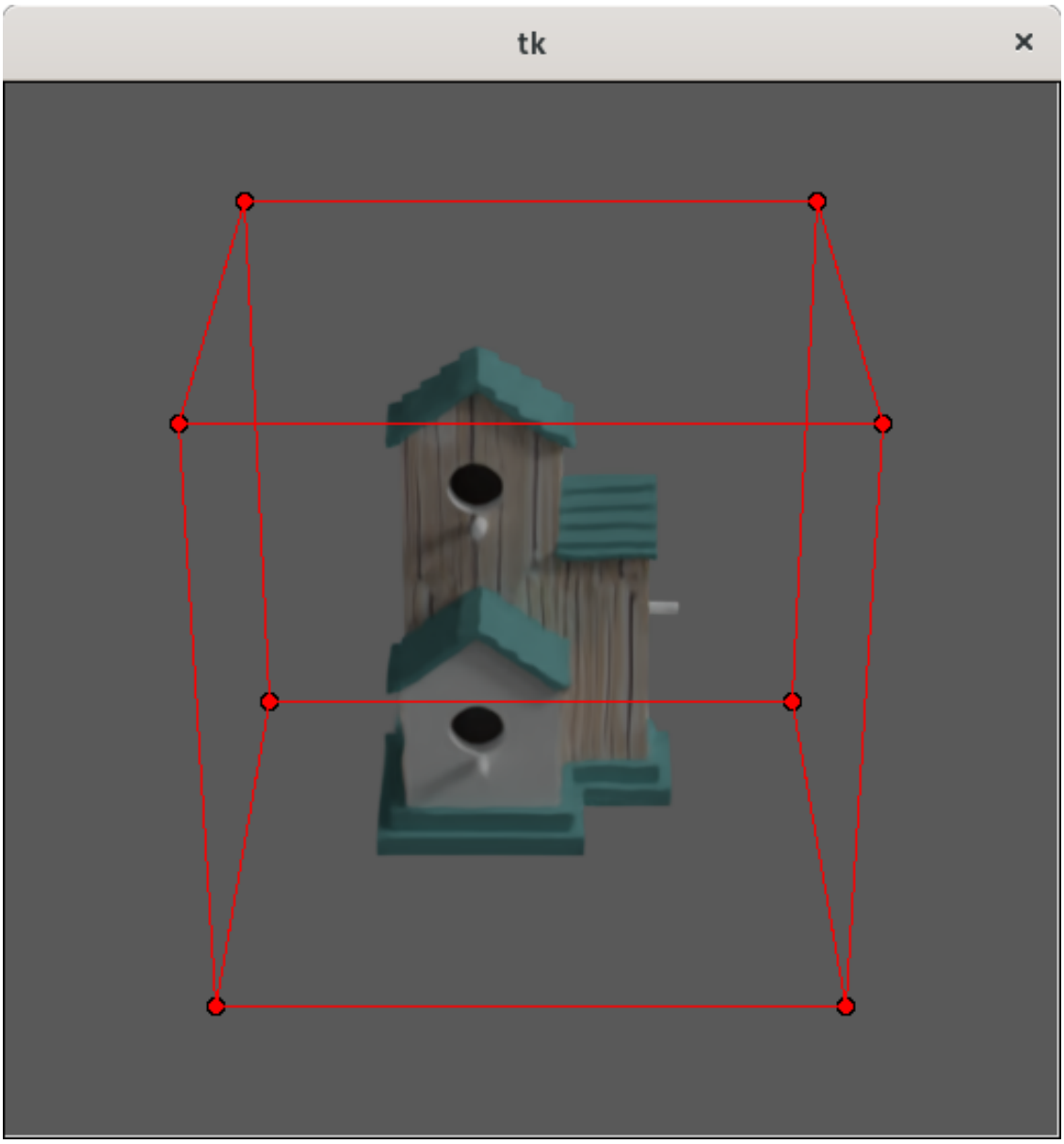


Figure 6.1: A screenshot of the real-time viewer running interactively on a laptop with only 2GB of Video RAM.

Chapter 7

Conclusion

This work seeks to examine what makes a 2-part view interpolation model tick, how they work, and how to make them work better.

In investigating the dynamics of a 2-part neural rendering model, we observe that neither the structure of the object representation or the type of CNN used is of critical importance. The primary barrier to a successful view interpolation model appears to be its generalizability to new object views. We observe non-generalizability to cause a significant disparity between training and testing losses. Augmenting the training data can somewhat mitigate this effect because it increases the number of views passed to the rendering CNN which. The CNN, however, is still capable of overfitting to image patches resulting in worse testing quality relative to single-pixel methods such as NERF [10].

A generalized output network capable of representing several objects at once would open many neural rendering applications. Unfortunately, the lack of constraints on the learned object representation results in inconsistent object representations. The mapping from these object representations to images is difficult to generalize, causing image quality to decrease as the number of objects sharing an output network increases.

Finally, in examining how we can edit the learned object representation, we observed that our explicit opacity term allows edits to generalize better to new occlusion scenarios. This effect suggests that differentiable renderers that explicitly reflect physical laws rather than learning them will better generalize to new situations.

Bibliography

[1] Vincent Sitzmann, Justus Thies, Felix Heide, Matthias Nießner, Gordon Wetzstein, and Michael Zollhofer. Deepvoxels: Learning persistent 3d feature embeddings. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2437–2446, 2019.

[2] Thu Nguyen-Phuoc, Chuan Li, Lucas Theis, Christian Richardt, and Yong-Liang Yang. Hologan: Unsupervised learning of 3d representations from natural images. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 7588–7597, 2019.

[3] Stephen Lombardi, Tomas Simon, Jason Saragih, Gabriel Schwartz, Andreas Lehrmann, and Yaser Sheikh. Neural volumes: Learning dynamic renderable volumes from images. *ACM Transactions on Graphics (TOG)*, 38(4):65, 2019.

[4] Thu H Nguyen-Phuoc, Chuan Li, Stephen Balaban, and Yongliang Yang. Rendernet: A deep convolutional network for differentiable rendering from 3d shapes. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 7891–7901. Curran Associates, Inc., 2018.

[5] Matthew M Loper and Michael J Black. Opendr: An approximate differentiable renderer. In *European Conference on Computer Vision*, pages 154–169. Springer, 2014.

[6] Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. Differentiable monte carlo ray tracing through edge sampling. *ACM Transactions on Graphics (TOG)*, 37(6):1–11, 2018.

[7] Shichen Liu, Tianye Li, Weikai Chen, and Hao Li. Soft rasterizer: A differentiable renderer for image-based 3d reasoning. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 7708–7717, 2019.

[8] Jun-Yan Zhu, Zhoutong Zhang, Chengkai Zhang, Jiajun Wu, Antonio Torralba, Josh Tenenbaum, and Bill Freeman. Visual object networks: Image generation with disentangled 3d representations. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 118–129. Curran Associates, Inc., 2018.

[9] Justus Thies, Michael Zollhöfer, and Matthias Nießner. Deferred neural rendering: Image synthesis using neural textures. *ACM Transactions on Graphics (TOG)*, 38(4):1–12, 2019.

[10] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *arXiv preprint arXiv:2003.08934*, 2020.

[11] Vincent Sitzmann, Michael Zollhoefer, and Gordon Wetzstein. Scene representation networks: Continuous 3d-structure-aware neural scene representations. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 1121–1132. Curran Associates, Inc., 2019.

[12] Angel X. Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. ShapeNet: An Information-Rich 3D Model Repository. Technical Report arXiv:1512.03012 [cs.GR], Stanford University — Princeton University — Toyota Technological Institute at Chicago, 2015.

[13] Abe Davis, Marc Levoy, and Fredo Durand. Unstructured light fields. In *Computer Graphics Forum*, volume 31, pages 305–314. Wiley Online Library, 2012.

[14] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation, Stichting Blender Foundation, Amsterdam, 2020.

[15] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[16] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[17] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[18] Christian Ledig, Lucas Theis, Ferenc Huszár, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, et al. Photo-realistic single image super-resolution using a generative adversarial network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4681–4690, 2017.