Neural Rendering for Transparent Objects

Patrick Radner
Technical University of Munich

Justus Thies
Technical University of Munich

Matthias Nießner
Technical University of Munich

Abstract

Neural rendering combines classical rendering techniques with learnable elements. It has shown great potential in areas, where the geometry of an object cannot be obtained at a fine enough resolution for the object to be re-rendered using the classical pipeline. Neural rendering has also great success generating images from novel view points.

In this work we apply the deferred neural rendering technique to transparent objects and complex scenes. We use a classical algorithm known as depth peeling. This algorithm renders the scene multiple times and during each iteration stores and removes the nearest surface from the scene, by using a depth mask. For each depth peeling pass, we compute per-pixel UV-coordinates and use them to look up a neural texture, as is done in [18]. We then stack all these layers and pass them through a neural network to obtain a final image. This allows us to see through transparent objects, where the term "transparent" also includes objects with holes, that are covered by a simplified geometry. It also complicates the original deferred neural rendering problem, as our network not only has to learn a rendering function, but also how to perform a blending operation on multiple objects.

1. Introduction

Although 3D-reconstructions have become quite accurate in recent times, they are still not good enough to generate photo-realistic re-renderings or novel views using only the classical rendering pipeline. Additionally many reconstruction techniques assume objects to be opaque and diffuse. Transparent objects tend to violate both those assumptions, which causes the quality of their reconstruction to suffer severely, if not break completely. Except for very constrained lab-setups there has not been much success in capturing a transparent objects geometry and texture. Furthermore, very fine topological details are usually reconstructed very poorly, as either holes end up being filled, or the objects end up being "torn apart". We show, that such objects can be rendered by using a conservative proxy and rendering it, as if it were a transparent object.

Neural networks, in particular generative adversarial networks (GANs), have been shown to be quite exceptional at novel view synthesis. [18] introduced an approach called *Deferred Neural Rendering*. In their approach latent feature vectors are sampled from neural texture maps, similar to how textures would be sampled in the classical rendering pipeline. A rendering network is then used to transform the sampled textures into the final image. Using this approach, photo-realistic images can be generated, even if the objects geometry is not all too accurate.

In this work we try to extend the idea of Deferred Neural Rendering to consider transparent objects, as-well as coarse geometries, that filled up holes form the original mesh or occlude objects that would be visible using a more accurate geometry. To this end, we do not only consider the first hit surface during rendering, but all the fragments that would be generated along view ray. These are computed using a classical rendering approach known as depth peeling. We then sample the fragments textures analogue to the original approach. For the rendering network we propose using a per-pixel multi-layer perceptron, instead of a U-net used in [18], as we have found it to produce sharper results and and rely more on the neural textures, which is important for scene editing and novel view synthesis from extreme view points.

2. Related Work

2.1. Image Synthesis

The recent advances deep learning allow us to use neural networks to generate high-quality, photo realistic images. In particular generative adversarial networks (GANs) [2] have been shown to provide exceptional results and can be found in most state of the art solutions [8, 6, 16, 18]. An important modification to classical GANs are conditional GANs (cGAN) introduced by [12]. In cGANs the authors propose

using conditioning parameters, which are fed into both the generator and the discriminator and allow to control the output of the generator. Using cGANs a common generator architecture is the U-net [14]: an encoder-decoder architecture with skip connections between same resolution encoder and decoder blocks.

2.2. Novel View Generation using Neural Networks

Neural rendering describes approaches using neural networks to predict, what a camera would see from an arbitrary viewpoint [19].

This paper extends the *Deferred Neural Rendering* approach introduced by [18], where the authors proposed augmenting the existing rendering pipeline with learnable elements; namely neural textures and a neural rendering network. [18] can generate high quality novel views, even if an object is approximated by a very coarse geometry. The authors also show, that their approach can be used for photo-realistic scene editing.

Other neural rendering approaches include for example [16], where they store their implicit scene representation in a 3D voxel-grid or the follow-up work [17], in which the voxel-grid is replaced by a continuous function represented by a neural network. Generative Query Networks (GQN) allow the generation of new objects by combining features from known objects [10]. [19] extends GQN by using attention along the epipolar line to model non-local dependencies.

2.3. Reconstruction of Transparent Objects

Reconstructing the geometry of transparent objects is a challenging tasks, as they violate most assumptions of standard algorithms. Modern 3D-reconstruction approaches often use RGB-D images to directly capture the geometry of an object, however, due to refraction, active depth sensors will fail to capture transparent objects. [1] proposes using the defects caused by capturing transparent objects with RGB-D cameras to detect and reconstruct them. A high quality reconstruction can only be achieved using very constrained setups [20]. Very recently ClearGrasp was introduced [15]. The paper uses deep learning to detect and transparent objects and fairly accurately predict their geometry in a RGB-D frame.

3. Method

3.1. Depth Peeling

In this section we briefly recap the depth peeling approach described in [11]. In computer graphics, depth peeling is a simple approach to achieve correct rendering order of transparent objects, without having to sort objects or pixel fragments. The algorithm works by utilizing 2 depth buffers. During each iteration the depth test is performed

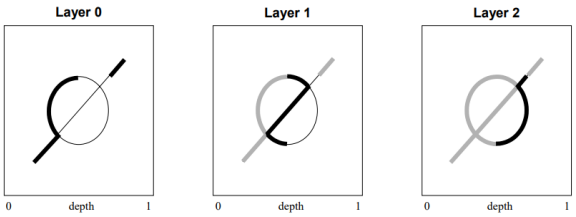

Figure 1: Illustration of the depth peeling algorithm. The bold segments are generated at each iteration. Grayed out segments are discarded by the depth test.

using the depth buffer from the previous iteration (initialized as 0) and the depth is written to the second buffer. This ends up "peeling" of the closest fragment, that has not been captured by previous iterations. The fragments generated this way are then usually blended using front-to-back α-blending. For our purpose we output the UV-coordinates (and object id for synthetic data) per pixel and pass them to the deferred neural rendering pipeline.

The number of depth-peeled input texture layers depends on scene complexity and was chosen conservatively, so no information would be lost. For most practical applications 4 to 8 layers should be sufficient, as anything beyond that is most likely occluded.

4. Deferred Neural Rendering

Deferred neural rendering replaces parts of the classical rendering pipeline with learnable elements [18]. In particular textures, which in computer graphics contain information such as color, opacity, normals and material properties are replaced by neural textures, which, for each texel contain a set of latent features. And the fragment stage is replaced by a rendering network, which takes the sampled features and predicts the final image. In our work we extend the pipeline introduced in [18], by having the rendering network not only cover the fragment stage, but also the output merger stage of the classical rendering pipeline.

We solve the task of novel view synthesis, given (approximate) 3D scene geometry with valid UV-mapping and pairs of camera position and RGB frames. At first we follow the classical rendering pipeline using depth peeling (see subsection 3.1) to generate layers of UV-coordinates. In order to separate the static rendering pipeline from the learnable part screen space UV-coordinates are precomputed and saved per depth peeling layer. They are then passed to our neural rendering setup. Analogue to classical rendering we sample our neural texture using bi-linear interpolation. This is done for each UV-layer separately. Sampled textures are then concatenated along the feature dimension and input into the rendering network. The learnable part of our rendering pipeline is fully differentiable and can be trained

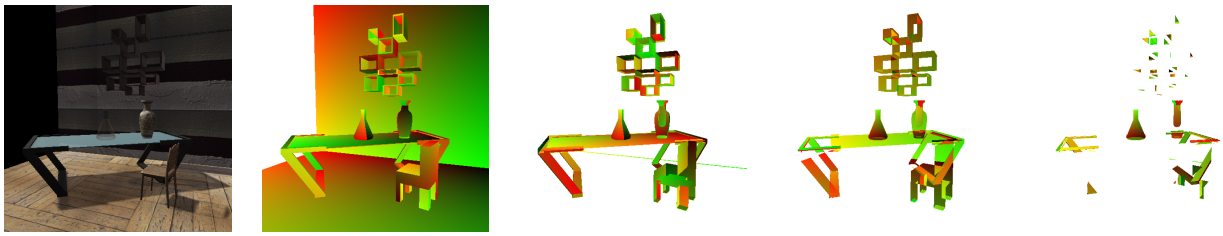

Figure 2: RGB frame and depth peeled UV-layers. Each object lives in its own UV-space, so we additionally render a segmentation mask per UV-frame. Note that the chair is approximated by a vary coarse proxy, simulating a bad reconstruction result. UV-coordinates are mapped to red and green channels. White indicates empty space and is mapped to 0 during sampling.

end-to-end. The entire problem can be formulated as optimization problem, where we try to find the optimal renderer $\mathcal{R}$ and texture T given scene geometry M and a training corpus $(I_k, p_k)_k^N$ consisting of camera poses p_k and images I_k. This leads to Equation 2:

$$T^*, \mathcal{R}^* = \underset{argmin}{T,\mathcal{R}} \sum_k^N \mathcal{L}(I_k, render(p_k, \mathcal{M}|T, \mathcal{R})) \quad (1)$$

Where $render$ describes the entire rendering pipeline. The classical rendering pipeline has no learnable parameters and we already precomputed the UV-coordinates $(uv)_i^L$, where L is the number of depth peeled layers. This allows us to shorten the problem to:

$$T^*, \mathcal{R}^* = \underset{argmin}{T,\mathcal{R}} \sum_k^N \mathcal{L}(I_k, Net(sample((uv)_i^L|T)|\mathcal{R})) \quad (2)$$

The function $sample$ only uses bi-linear interpolation in our case, but it can be extended to use hierarchical textures as shown in [18]. The function $\mathcal{L}$ denotes our training loss. In the following sections we will explain the components of our neural rendering pipeline in detail.

4.1. Neural Textures

In classical computer graphics textures are used to store detailed surface information, such as color, opacity, normals and material properties needed to compute accurate lighting. During rendering UV-coordinates are computed for each fragment, which are then used to look up the corresponding values in the texture map. Texture values are interpolated using bi-linear interpolation or tri-linear interpolation in the cast of mip-maps. Neural textures work in the same way: latent features are fetched from high-dimensional textures according to the computed UV-values and interpolated using bi-linear interpolation. This interpolation scheme is differentiable, which allows us to train our neural rendering pipeline end-to-end. [18] also implemented a way to use hierarchical textures, which help avoid minification and magnification problems. The significant difference here is, that in classical rendering the geometry needs to be highly detailed. Normal maps can be used to render some very fine surface details, without explicitly modeling them. In neural rendering we can also use coarse geometry proxies to sample our feature maps. The neural renderer can then generate fine-scale details.

4.2. Deferred Neural Renderer

We sample our screen space feature maps using the UV-coordinates obtained by rendering our geometry proxies with depth peeling. The classical rendering pipeline would now compute a lighting function to compute the final fragment color and blend the computed fragments together using $alpha$-blending. We instead stack the sampled screen space feature maps along the feature dimension and give them as input to our neural network. Additionally we can add the view direction from the camera to our surfaces, making it easier to compute view dependent effects, such as specular highlights. While rendering our geometry each fragments world position is stored in an auxiliary texture at the computed UV-coordinates. During training these position textures P are sampled and the view direction v is computed as the normalized vector from the camera position p_k to the surface position:

$$v = normalized(p_k - sample(uv|P)) \quad (3)$$

The neural rendering network is at the core of our pipeline. It takes as input the stacked screen space feature maps and predicts the final output image. To achieve this it has to learn a function replacing both the fragment shader and output merger stage of the classical rendering pipeline.

4.3. Network Architecture

[18] proposed using U-nets unet for rendering their screen space feature maps. For our work, however, these U-Nets seem to overfit the training data a lot and thus fail to predict novel views, when trying to extrapolate too far from the sample distribution. We also found, that when us-

ing very coarse geometry proxies, such as simple bounding boxes, U-Nets ended up mapping the view direction to the output directly and ignoring textures, which ended up causing severe artifacts when trying to edit the scene. Instead, we propose using a per-pixel multi layer perceptron (MLP), as is done in [17]. These seem to work better for the extended DNR problem of combining multiple layers of feature maps to form an output image. Per-pixel networks also come with the added benefit of being invariant to scaling, as they do not consider the neighborhood of the sampled features. The standard U-Net used in [18] uses about 16 million learnable parameters, whereas our per-pixel networks usually have about $40,000$. All of these parameters are applied to predict each final pixel from the sampled features, making it difficult for the networks to overfit and forcing the network learn a more general rendering function, while shifting the object-specific information into the neural textures. Per-pixel MLPs can be implemented very efficiently by building a network of only same convolutions with a kernel size of $(1, 1)$. As activation function we used LeakyReLU ($\alpha = 0.2$). For our textures we used 16 feature dimensions and the first layer of our MLP uses 256 hidden units. Two sets of linear layers and non-linearities form a block and the input of each block is added to its output. Afterwards a linear layer is inserted, in which the number of feature channels is halved. This is analogue to a ResNet [3] architecture using only 1×1 convolutions, when looking at how the network is applied to the entire image, not just one pixel. Our default network consists of three such blocks, which is results in about $34k$ parameters. For simpler scenes we were able to reduce the number of blocks to one with only a minor loss in quality. We used Batch Normalization [5] after each linear layer.

4.4. Training

As mentioned previously, our pipeline can be trained end-to-end. We precompute the UV-maps for each depth peeling layer and load them at training time. They are paired with the ground truth RGB image, as well as camera pose to form one data sample. This can be seen in Figure 2. The UV-maps of our synthetic datasets also contain a segmentation mask, as each objects has its separate UV-space, we thus use separate textures for each object. This does however not impose a limit to real world applicability, as the system will also work when simply considering the entire scene as one object, with one unified UV-space and one large neural texture. UV-maps and masks are used to select and sample the neural texture. The sampled features are finally put through the network to generate the output image, our loss function w.r.t. the ground truth RGB image is applied to. As our loss we use a combination of $L1$ and perceptual loss. Perceptual loss is based on the predicted features of a standard, image-net pretrained VGG network

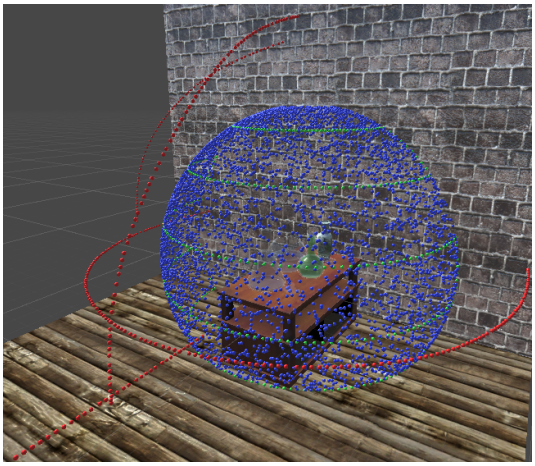

Figure 3: Visualisation of our training and test sets. Blue: training data; Green: Interpolating test set; Red: Extrapolating test set.

[7]. In our experiments we found, that perceptual loss helps or network to generate finer detailed textures and high frequency light effects (specular highlights). Using only perceptual loss, however, resulted in visibly wrong colors (e.g. green objects were suddenly blue). Adversarial training only delivered descent results after an initial training phase with a more stable loss and a very reduced learning rates. In our experiments we did not see any significant quality improvements form using adversarial loss.

As the per-pixel networks used in this paper do not consider neighboring information, we do not need to perform usual data augmentation steps like random cropping or scaling, which is done for the U-Nets used in [18]. We have however found, that randomly adding layers of zeroes can improve generalisation performance in simple scenes, especially when the order of seen objects does not change very often. These zero-layers are tensors of the same shape as our screen space feature maps and they are inserted aligned with our the sampled texture data. This seems to reduce correlation between textures meaning that the features stored in the neural textures only describe the corresponding surface and the network is force to learn something more similar to a blend function. Without this form of augmentation we have seen networks produce severe artifacts when trying to remove objects from the scene.

Our networks and textures are trained using stochastic gradient descent. In particular, we use the Adam optimizer [9] and the entire neural pipeline is implemented in PyTorch [13]. As parameters we use a learning rate of 0.001 and default Adam parameters, as suggested in [18]. For our experiments with adversarial loss we turned down the initial learning rate as low as 0.0004.

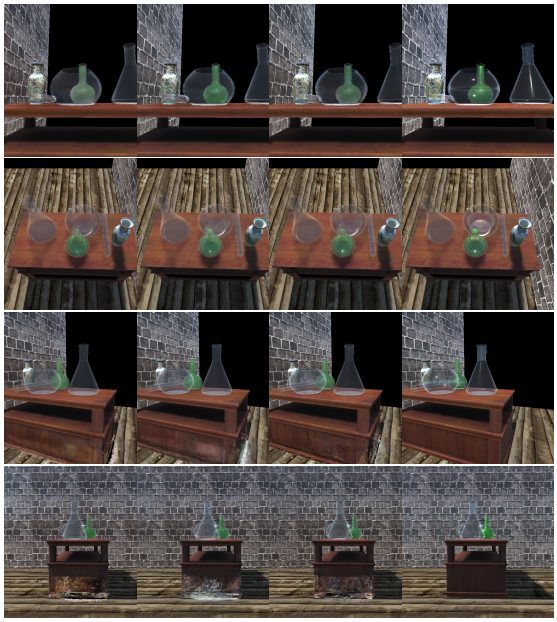

Figure 4: Samples using different extrinsics. Left to right: No extrinsics, 3D view direction, spherical harmonics, ground truth. The first two rows were taken from the interpolating test set, the last two from the extrapolating one. We can see that without extrinsics the renderer struggles to generate specular highlights on the interpolating test set. When moving to more extreme viewpoints neither approach manages to capture the highlights. Note that the bottom of the table was not directly visible in most training images and thus shows severe artifacts.

4.5. Training Data

All of our training data was generated synthetically using the Unity3D engine. For our default experiment setups the training data was sampled from a hemisphere with the camera looking at the center. The hemisphere usually had to be cropped, to avoid looking at the scene from below the floor. We used two test-sets: The interpolating test-sets take a path through the same hemisphere we sampled the training data from, whereas the extrapolating tests were generated by manually walking through the scenes. A visualisation of the training and test-sets can be seen in **??**. To show our performance given a poor reconstruction we performed experiments using either bounding boxes or simple, low-poly, box-like objects which were manually created using Blender. In all setups we used ground truth camera positions.

4.6. Extrinsic Parameters

The glossy nature of transparent materials, such as glass, leads to so-called specular highlights, which can be classified as high-frequency, view-dependent details and are thus very nasty to render. In order for our algorithm to handle such view-dependent effects we look at two different methods. The first method directly adds the per-pixel 3D view vector to the renderers input, whereas the second one uses the view vector to compute the first three bands of Spherical Harmonics and adds those as input to the rendering network, as was done in [18]. The process of computing the view vector was described in subsection 4.2. Note that these methods add respectively 3 and 9 additional input channels per depth-peel layer. Samples of our results can be seen in Figure 4 and the quantitative evaluation based on PSNR [4] is shown in Table 1. Our experiments showed, that adding extrinsic information can help with view-dependent effects, when generating images close to the training corpus. When synthesizing novel views from points far away from the training corpus, however, our method still cannot generate proper specular highlights.

	Interpolating	Extrapolating
No extrinsics	33.09	32.96
View vector	33.19	32.99
SH	33.80	32.52

Table 1: Ablation study of auxiliary renderer inputs (camera extrinsics), which are supposed to help with view-dependent effects including specular highlights. Values are given in dB and denote the PSNR. The approach using spherical harmonics performs better on the interpolating test, but does not generalize as well to more extreme view points.

5. Results

5.1. Baseline

In this section we show both qualitatively and quantitatively, that our solution outperforms existing approaches. Our results are evaluated with the commonly used Peak Signal to Noise Ratio (PSNR) [4]. There exists not much work in the area of reconstructing and re-rendering transparent objects. We thus compare our solution against the original Deferred Neural Rendering approach by [18], which was not meant to deal with transparent objects. Surprisingly, the network is still able to produce descent results. We assume this is due to the large capacity of the UNETs, allowing them to produce an image essentially as a function of view direction. We do however observe severe blurring when rendering transparent objects. Samples are shown in Figure 7. The comparison based on PSNR is shown in Table 2, where we can see DNRs quality diminishing when using the extrapolating dataset, which coincides with our over-fitting assumption.

We also compare our solution against DeepVoxels [16], as they accumulate values along a view ray, which at least

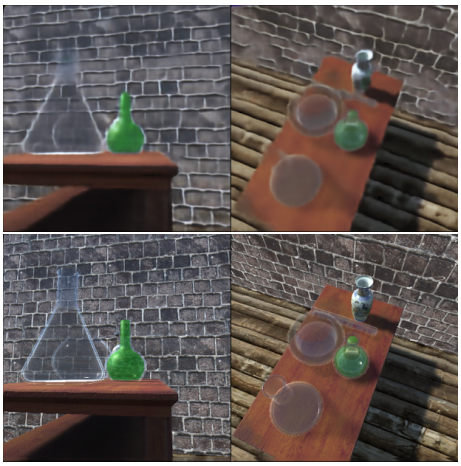

Figure 5: Training images from running DeepVoxels ([16]) on our dataset (top). Ground Truth (bottom).

	Interpolating	Extrapolating
DNR	33.52	30.86
Ours	33.19	32.99

Table 2: Comparison of Deferred Neural Rendering to our approach using PSNR. Values are given in dB.

in theory means, that the algorithm could be able to deal with transparency. In practice, however, we were only able to get very blurry and over-smoothed re-renderings of training images and the algorithm failed to even generalize to our interpolating test set. These results were achieved after disabling adversarial loss and only training on L1 loss with a low learning rate $(1e-5)$, anything else lead to a total collapse. Even with these measures not every training run delivered successful results. We had to stick with the original 32^3 voxel resolution, as DeepVoxels is very memory intensive, this might also be a reason as for why the algorithm was not able to deal with our complex scene. Note that DeepVoxels solves the more general problem of novel view synthesis from only images and poses, whereas we assume geometry to also be known. We thus conclude, that DeepVoxels, same as a lot of conventional reconstruction algorithms, is not generally able to handle complex scenes containing transparent objects. Our results when trying to use DeepVoxels can be seen in Figure 5.

5.2. Geometry Proxies

All our experiments were performed using synthetic data, since there were no publicly available real-world datasets of transparent objects at the time. Analogue to [18] we tried using coarse geometry proxies to render our objects. This tests the robustness of our algorithm wrt. geometry and suggests, that it might also perform well in a real-world environment. Our geometry proxies were created manually using the 3D modelling tool Blender and are much worse than what can be expected from state-of-the-art reconstruction algorithms. Figure 1 gives an idea of such a proxy. In Figure 6 the chairs geometry is approximated using a very blocky structure, which can be inferred from the first two uv-maps.

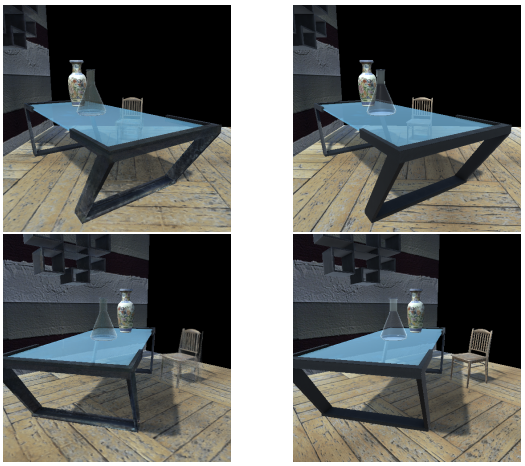

Figure 6: Results when using coarse geometry proxies. Left: Our results; Right: Ground truth

[18] goes as far as using simple boxes as geometry proxies to show the robustness of their algorithm. We tried to also apply this idea to, highly complex geometries, which pose a nightmare even for modern reconstruction algorithms. In particular we looked at plants and trees and tried to approximate them using simple shapes. While the results are not good enough for practical use, in our opinion they are quite good when considering the setup: "Draw a plant from an ellipsoid". When looking at the rendered test set video, transitions between different faces of a proxy cube caused difficulties for our renderer. We thus decided to use ellipsoidal proxies instead, which drastically improved multi-view consistency. Figure 8 shows some samples of a complex scene, where complex objects were approximated using only such extreme proxies. These images were created using adversarial loss after initialisation with L1 loss and a texture resolution of 256^2 with 64 feature channels, to accommodate for the increased object difficulty.

6. Limitations

By replacing the U-net renderer of [18] with a per pixel network our solution is no longer capable of in-painting areas where little to no surface information is present in the training set. Figure 4 and Figure 8 both show, rendering unseen regions leads to severe errors. While completely unseen areas, such as the ones Figure 8 can of course never be re-rendered by any approach, regions like the bottom of

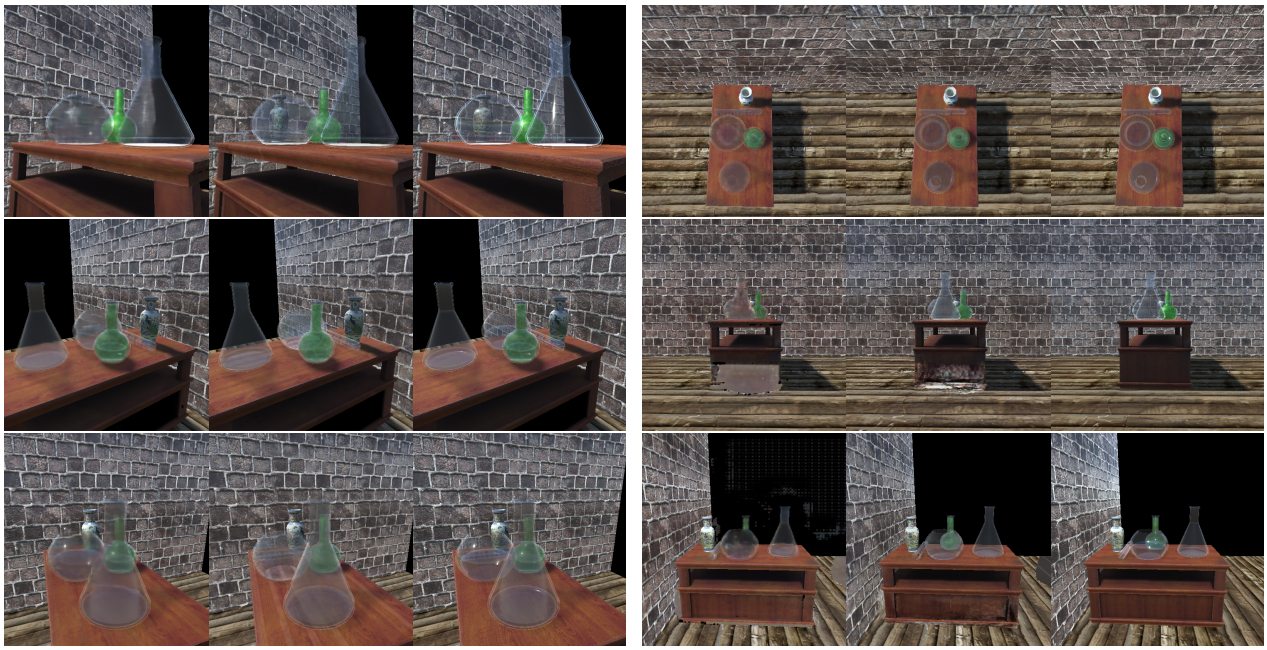

Figure 7: Comparison of our results to Deferred Neural Rendering by [18]. Images order (left to right): DNR, ours, ground truth. Left column was taken from the interpolating dataset, right from the extrapolating.

the table in Figure 4 could at least produce less striking errors, when using an in-painting solution. As a possible remedy for this we could make use of adversarial loss for novel views, as was done in [16]. Using this approach we would perform in-paining by using the gradients of a discriminator network to predict unseen areas.

Another limitation is the reliable generation of specular highlights. Our results in subsection 4.6 show, that we cannot generate these effects when rendering novel views from arbitrary positions.

7. Conclusion

We introduced an extension to *Deferred Neural Rendering* by [18] that allows us to generate novel views of scenes containing transparent objects. The same solution can also applied to objects with holes that have been filled by an imprecise reconstruction algorithm. We compared out per-pixel MLP networks to the U-Nets used in *Deferred Neural Rendering* and provided theoretical reasons and empirical evidence, why we think per-pixel networks should be used for this kind of neural rendering approach. Our solution is capable of generating realistic novel views of scenes with accurate geometry, and it can also generate results when only a coarse geometric scene representation is present. Neural rendering is a very powerful tool and the community is just beginning to discover its potential. We think our approach to rendering highly complex objects using primitive geometry proxies can be improved to also generate photo-realistic results, allowing us mostly to skip 3D reconstruction.

8. Future Work

There are many avenues left for the neural rendering community to explore. *Deferred Neural Rendering* in our opinion shows great potential and should be developed further. We did some minor experiments in the direction of super-resolution, the process of generating higher resolution images from a set of lower resolution inputs. In theory, using our neural textures and just sampling at higher resolution should be able to produce results, at least as good as the low resolution images. Our limited experiments, however, showed some severe artifacts, so more research in this area is required.

In this paper we only worked on synthetic data. With the introduction of ClearGrasp [15] there is now a tool that can produce reasonably good 3D-reconstructions of transparent objects. We did not have time to implement this approach in our pipeline. We strongly believe, that a combination of the high quality reconstructions obtained with ClearGrasp our solution can be used to generate photo-realistic novel view renderings in in-the-wild scenarios.

Using the precomputed UV-maps actually turned out to be fairly constraining, as they need to be stored using high accuracy (standard 8-bit color is not enough) and thus they also need a lot of disk space and loading time. We suggest, that for future experiments it might be beneficial to

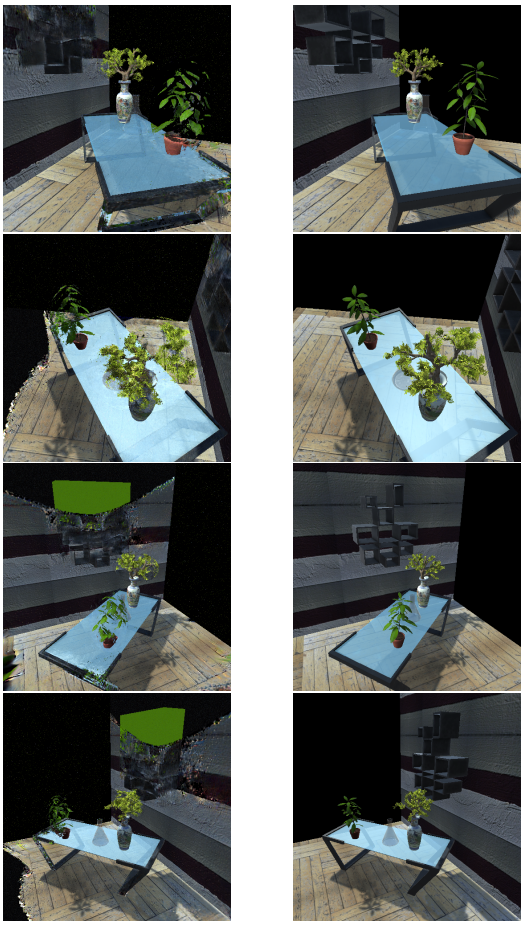

Figure 8: Rendering highly complex objects using primitive geometry proxies. Plants were approximated using ellipsoids; the shelf in the background is approximated using a simple box. Note, that the test images were taken from extreme view-points. The training set was again sampled from a hemisphere in front of the table. Left: Our results; Right: Ground truth.

render the UV-maps online using a GPU based renderer. This might speed up the training and inference process significantly.

9. Acknowledgement

Our synthetic datasets were generated using the Unity3D engine. The 3D models and textures used came from various artists that made their work freely available on the unity asset store. Some additional assets were also imported from *www.artec3d.com*. Any geometry proxies more complex than a bounding box were created using Blender.

References

[1] Nicolas Alt, Patrick Rives, and Eckehard Steinbach. Reconstruction of transparent objects in unstructured scenes with a depth camera. pages 4131–4135, 09 2013.

[2] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[4] A. Horé and D. Ziou. Image quality metrics: Psnr vs. ssim. In *2010 20th International Conference on Pattern Recognition*, pages 2366–2369, 2010.

[5] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[6] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks. *CoRR*, abs/1611.07004, 2016.

[7] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. *Lecture Notes in Computer Science*, page 694–711, 2016.

[8] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. *CoRR*, abs/1812.04948, 2018.

[9] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[10] Ananya Kumar, S. M. Ali Eslami, Danilo J. Rezende, Marta Garnelo, Fabio Viola, Edward Lockhart, and Murray Shanahan. Consistent generative query networks, 2018.

[11] Bavoil Louis and Myers Kevin. Order independent transparency with dual depth peeling. 02 2008.

[12] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets, 2014.

[13] 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.

[14] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015.

[15] Shreeyak S. Sajjan, Matthew Moore, Mike Pan, Ganesh Nagaraja, Johnny Lee, Andy Zeng, and Shuran Song. Clear-grasp: 3d shape estimation of transparent objects for manipulation, 2019.

[16] Vincent Sitzmann, Justus Thies, Felix Heide, Matthias Nießner, Gordon Wetzstein, and Michael Zollhöfer. Deepvoxels: Learning persistent 3d feature embeddings. In *Proc. Computer Vision and Pattern Recognition (CVPR), IEEE*, 2019.

[17] Vincent Sitzmann, Michael Zollhöfer, and Gordon Wetzstein. Scene representation networks: Continuous 3d-structure-aware neural scene representations. In *Advances in Neural Information Processing Systems*, 2019.

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

[19] Josh Tobin, OpenAI Robotics, and Pieter Abbeel. Geometry-aware neural rendering, 2019.

[20] Bojian Wu, Yang Zhou, Yiming Qian, Minglun Cong, and Hui Huang. Full 3d reconstruction of transparent objects. *ACM Transactions on Graphics*, 37(4):1–11, Jul 2018.