Real-time Neural Human Rendering in VR

Long Cheng Weirong Chen Tianfu Wang Markus Pobitzer
ETH Zürich
{locheng, weirchen, tianfwang, pobmarku}@ethz.ch

Supervised by Dr. Sergey Prokudin

Abstract

We propose a point-based neural rendering pipeline developed for real time applications in the virtual reality (VR). We use point clouds as our representation of the human models and propose an adaptive shader to render the point clouds in Unity. Through Unity's neural network inference library we pass the rendered images through a neural network to enhance the output. The main focus lies in making the pipeline as compact as possible such that real time interactions in VR are possible. Our experiments show that our framework can interactively (∼ 15 FPS) render both static and dynamic human pointcloud with good rendering quality and consistency.

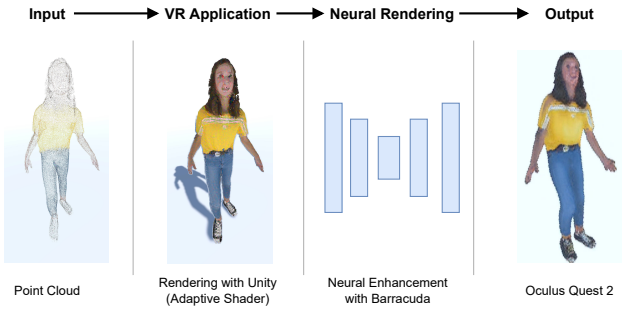


Figure 1. Point-based neural human rendering pipeline for VR

1. Introduction

Generating photorealistic and high-quality imagery for humans in an interactive time frame is a problem that has attracted much interest and development from both the computer graphics and computer vision community [24, 25]. Successfully tackling this problem can enable exciting applications, such as immersive virtual gaming and entertainment experiences and interactive telepresence. However, rendering photorealistic humans in real-time faces many challenges. Firstly, achieving photorealism for humans is particularly challenging due to the "uncanny valley" effect [12], where even small distinctions in the generated imagery can cause negative perceptual evaluation when rendering humans compared to other objects. Secondly, photorealistic real-time rendering is a challenging problem in computer graphics. Traditional methods rely on explicit, often physically-based, modeling for geometry, material, and light transport in a scene. [26, 46]. These methods require high-quality geometry model information, such as high-resolution mesh and texture maps. Additionally, directly solving for light transport and global illumination using an explicit modeling algorithm such as ray tracing is slow to converge and computationally intensive, let alone capable of real-time applications.

Recently, the new approach of using neural rendering in rendering photorealistic imagery, including humans, has been marked by tremendous progress [38, 39]. Neural rendering uses a data-driven approach to model scenes from high-quality image samples. Current work in neural rendering has already achieved photorealism in areas challenging for traditional graphics pipelines. In the realm of photorealistic human rendering, SIMPLpix [30] builds on pix2pix [16, 43], a generative adversarial network (GAN) [13] driven image-to-image translation model, to achieve realistic texture coloring of the human body, while maintaining the flexibility and user controllability of using an explicit 3D human mesh. Additionally, there have been developed for point cloud based neural rendering models [4] that demonstrated great detail, photorealistic results of human rendering in real-time. Compared to the traditional mesh 3D representation, point cloud data does not require surface estimation and can be easily obtained from consumer-grade 3D scanners. It is also shown that point cloud based approaches can achieve compelling results on thin regions of the 3D object, such as foliage and hair. The work LookinGood [24] also uses a neural network to refine initial data obtained from the low-resolution point cloud and texture data in real-time to achieve realistic human rendering. These promising developments lead us in the direction

of combining the use of a point cloud representation of a human with leveraging deep neural architectures as a refinement module.

Our project aims to investigate the efficacy of rendering virtual 3D humans interactively in a VR setting. Specifically, we would like to leverage point cloud based methods for our human geometry representation and rendering model. Our pipeline takes a human model's sparse 3D point cloud and visualizes it in a virtual space. We then use neural rendering as an image-to-image enhancement module to generate a refined human rendering as our output. We leverage Unity, a popular development platform for graphics and VR applications, to set up our virtual scenes and run our neural rendering models as a post-processing pipeline. We leverage current point cloud rendering libraries in Unity (Pcx) and enable an adaptive point cloud rendering strategy using our custom geometry shader. We designed our network architecture to be lightweight [3] while taking special care of our objective function design and data generation pipeline. These decisions allow us to render virtual humans with good quality and consistency. We tested our framework on various input data, including static humans, dynamic humans, and scenes with multiple humans. Our results in VR are promising in terms of quality, consistency, and speed. Furthermore, our pipeline can visualize the rendering results interactively ($\sim$ 15 FPS) in an Oculus Quest 2 VR headset.

2. Related Work

2.1. Point-based Rendering

There has been a long history in computer graphics of using point-based primitives as 3D representation [22]. Compared to other traditional geometry primitives, such as triangle meshes, point-based representations do not encode connectivity or adjacency of the 3D points. Although this lack of surface information can lead to challenges in the rendering process, it also makes point-based primitives extremely flexible and, more importantly, friendly to data captured from modern 3D scanning devices. Therefore, point-based primitives are very useful for real-life interactive applications. Over the years, numerous developments in point cloud data structure and rendering techniques, such as surface splatting, have been proposed to improve the rendering quality and efficiency of point cloud data [28, 49]. Recently, there has been increasing interest in using deep neural architectures to achieve photorealistic rendering of point-based scenes. Aliev *et al*. [4] encodes the local geometry and appearance of points using learnable neural descriptors and achieves realistic results from commodity RGB-D scanning devices. There is also development on leveraging point cloud data obtained from structure from motion pipelines and taking advantage of high dimensional SIFT features for realistic rendering [29].

2.2. Image-to-image Neural Translation

From a computer graphics viewpoint, achieving different rendering effects requires complex scene modeling with high-fidelity meshes, realistic light sources, physical-based materials, etc [6]. Modern rendering techniques such as ray tracing can provide more photorealistic results than the traditional pipeline, while they are computationally heavy and hard to achieve real-time performance without a powerful GPU. On the other hand, from a computer vision viewpoint, one can bypass the scene modeling step and try to solve the rendering problem as a pure learning-based approach, which directly performs 2D-to-2D image translation through deep neural networks. For instance, semantic photosynthesis takes the input of a semantic map and generates the photorealistic image accordingly [7, 16, 44], which model the synthesis process using Convolutional Neural Networks (CNNs) with the GAN loss. Another task that leverages the image-to-image translation is the style transfer, which takes a raw input image and converts it to the given style. Early methods train one single network for transferring a specific scale [11, 40, 41], while more recent works are capable of targeting multiple styles in a single model [8, 23].

2.3. Real-time Rendering in VR

Unlike normal neural network, real-time neural network has to be designed specifically to optimize run-time speed for user applications. In order to run neural network on mobile devices which possess limited computing power, MobileNetV2 [33] proposed mobile nets with inverted residual connection and depth-wise separable convolutions (3x3 convolutions followed by 1x1 point-wise convolution). However, during implementation, the repeating Relu6 layer in the building block is not supported by Barracuda [37] , making real-time inferring using MobileNetV2 hard to be implemented on VR devices natively. Other neural network architectures on edge devices include: MobileNetV3 [15] FBNet [47], and EfficientNet [35].

Virtual Reality (VR) has gained attention in various fields in the past few decades. Using VR device, humans can maximize their immersive experience through their eyes in the virtual world. At the same time, the extremely high requirement for display resolution, refresh rate, and stereoscopic images rendering pose challenges for VR to permeate the industry and consumer market. To solve the rendering issue in VR, foveated rendering methods using eye tracking like [14] are gaining popularity. Additionally, Research using neural networks to get decoded foveated images relieves the latency issue in VR rendering [19]. For specific tasks of rendering in VR, the key point lies in clean and efficient code design and architecture.

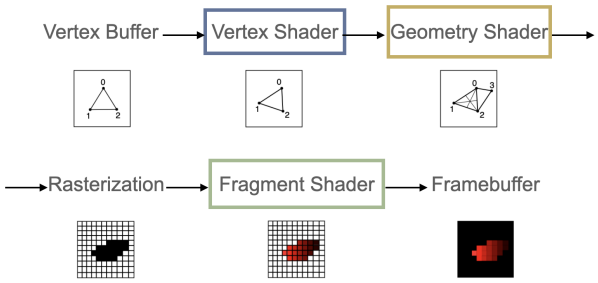


Figure 2. Graphics pipeline. *Source*: Vulkan tutorial [27]

3. Method

3.1. Adaptive Shader

Graphics rendering pipeline in Unity To construct an efficient pipeline for rendering, it is essential to understand the graphics pipeline in Unity. For a standard graphical task, the CPU firstly makes a draw call to the GPU. Afterwards the vertex shader gets called. In vertex shader, vertex information, including local position, UV coordinate, color, and normal, are transformed from local coordinate system to world coordinate system. Those become the input for the fragment shader. In fragment shader, shading information for fragments is calculated based on the color and normal information. Shading tests and z-buffering tests are used to render visibility of vertices and faces.

Geometry shader is an optional shader that generates additional geometric primitives between vertex shader and fragment shader. In the scenario of rendering humans in VR, observers are allowed to freely explore the scene. At close distance, point clouds become sparse and sometimes hard to recognize. As a result, preprocessing of point clouds is necessary. Geometry shader takes as input a set of vertices and outputs a set of new primitives. The benefits of geometry shader are: With the help of GPU, geometry shader is very efficient in processing vertices in graphics pipeline. And it is easy to control and intuitive to modify the parameters of the geometry shader. With geometry shader, the full graphics pipeline before entering neural rendering is shown in Figure 2.

In our case, each point in the point cloud is reshaped as a square facing its normal. In geometry shader, the point size can be controlled adaptively to decrease visual artifacts. Intuitively, the size of a point can be set inversely proportional to the distance between camera and object point:

$$\text{size}_p = \frac{\lambda}{\text{dist}_{cp}} \tag{1}$$

in which p is the independent point in the point cloud, dist_{cp} is camera-to-point distance. However, this relation might not satisfy all distances needed. To figure out the best relationship between size of point cloud and camera-to-point distance, human-judged best point sizes at different distances are recorded. Those one-to-one mappings are fitted to a polynomial and clamped in a range. The final handcrafted relationship between point cloud size and camera-to-point distance is as follows:

$$\text{dist}_{cp} = \text{clamp}(\text{dist}_{cp}, a, b) \tag{2}$$

$$\text{size}_p = c + d \cdot \text{dist}_{cp} + e \cdot \text{dist}_{cp}^2 \tag{3}$$

a,b indicate the lower bound and upper bound for clamping. c,d,e are derived from the fitted polynomial. This serves as the most important function in determining the final point size.

Algorithm 1 Forward Pass

for each vertex in Vertex Buffer **do**
  Get point size based on camera-to-object distance.
  Obtain the normal of this vertex in world coordinate.
  Get the vertices of the square based on camera-to-object distance and world normal.
end for
for each fragment in Fragment Buffer **do**
  Get Ambient light strength from light source.
  Get Specular parameter based on worldNormal and halfVector.
  Get Diffuse parameter from worldNormal worldLighrDirection.
  return Ambient + Specular + Diffuse
end for

Lastly in graphics pipeline, a view-dependent shadow effect is achieved through Blinn-Phong shading [5] and one additional shadow pass. The source code for shadow pass is from Pcx. These effects visualize shadows and occlusion in the model. The pseudocode of the complete forward pass is in 1

3.2. Neural Rendering

3.2.1 Data generation

We use the PyTorch3D library [31] to generate our datasets. PyTorch3D is an Our goal is to take sparse point cloud input and render dense, high-quality output through our neural network. In addition, we would also like the human color to be consistent when viewed from different depths. To address this, we use our custom PyTorch3D adaptive point cloud shader to render training and target images. The implementation of the PyTorch3D shader follows the same as described in the previous geometry shader section. Specifically, we also use equation (1) to determine our point size. Our PyTorch3D shader is efficient since it leverages the PyTorch backend, enabling batched processing on multiple images and camera poses through the GPU.

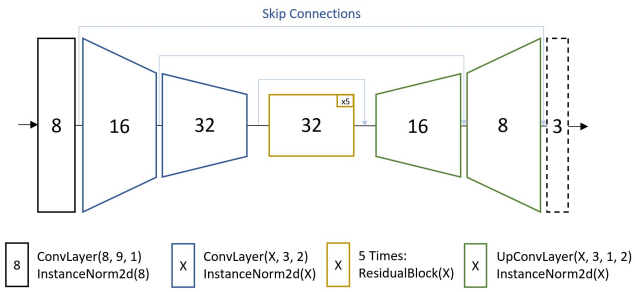


Figure 3. Overview of our model. The numbers in the shapes indicate the number of output channels we tried to keep them as small as possible to achieve real time performance. Blue arrows indicate Skip connections, differently colored shapes indicate different operations.

3.2.2 Network Architecture

The choice of the network was limited through the supported operations in Barracuda [37] and an export of the model to ONNX opset version 9 [36]. Our network is based on [17] using Instance Normalization as introduced in [42] and the used implementation can be found under [3]. The architecture is fully supported by Barracuda. Given the architecture we reduced the channel numbers to make the network compacter, an overview can be seen in Figure 3.

The network follows a U-Net [32] style architecture, with the goal of learning as many relevant features in the image as possible. An in depth description can be found in Table 1.

For training, we use 5880 image pairs of three different people rendered from different depths, elevations and azimuth angles. We use Adam [20] as our optimizer.

3.2.3 Loss

The output of the network I_X should yield a perceptually similar result to the ground truth I_{gt}. For this we mainly rely on losses that are based on features of pretrained networks on ImageNet [10] as these come close to the perception of humans as discussed in [18, 48]. Our used loss is a combination of a Learned Perceptual Image Patch Similarity (LPIPS) [48] and a feature based distance on a pretrained VGG-16 [34] network as introduced in [18].

For the LPIPS part, we use the provided implementation of the authors [48]. The variant we use is based on the AlexNet [21]. The second part is based on a pretrained VGG-16 network. The loss directly follows from SMPLpix and minimizes the $L1$ loss between VGG activations [30]. The final loss is just a combination of the two:

$$L = \lambda_1 \, L_{LPIPS}(I_{gt}, I_X) + \lambda_2 \, L_{VGG}(I_{gt}, I_X).$$

Layer	Filters	Kernel	Stride	Out
Input Image	-	-	-	I
ConvLayer(I)	8	9	1	Y1
ConvLayer(Y1)	16	3	2	Y2
ConvLayer(Y2)	32	3	2	Y3
Residual Block(Y3)	32	3	1	X
UpLayer(X + Y3)	16	3	1	X
UpLayer(X + Y2)	8	3	1	X
Refl. Padd(X + Y1)	-	-	-	X
Convolution(X)	3	9 x 9	1 x 1	Y
Description of Custom Layers				
ConvLayer(X)	F	K	S	
Refl. Padd(X)	-	-	-	X
Convolution(X)	F	K x K	S x S	X
Instance Norm(X)	F	-	-	X
ReLU(X)	-	-	-	Y
UpLayer(X)	F	K	S	
Interpolate(X)	-	-	-	X
ConvLayer(X)	F	K	S	Y

Table 1. In detail description of the architecture. The input image has format H x W x 3, same as the generated output. The Reflection Padding (Refl. Padd) has as padding parameter K // 2, where K is the kernel size of the following Convolution and // represents integer division. The Residual Block gets repeated five times.

The parameters λ_1 and λ_2 indicate the importance between the losses, we set $\lambda_1 = \lambda_2 = 1$.

We also looked at the VGG-19 based loss used in Neural Point-Based Graphics (NPBG) [4]. However this did not yield better results as we will discuss in Section 4.

3.3. Framework

This project aims to develop a full pipeline for rendering photo-realistic humans in VR devices from a sparse point cloud representation, allowing interactive real-time user experience from novel view points. Such a pipeline requires the integration of the game engine, neural inference package, and the VR hardware altogether. We choose Unity engine as our development platform to build the VR application, which provides rich support for 3D and VR projects. We extend the existing point cloud rendering shader Pcx with the adaptive point size function for the Windows platform. To introduce neural network inference into Unity, we leverage the Barracuda package [37] from Unity-Technologies to perform neural rendering via a pretrained network. Oculus Quest 2 is selected as a deployment device which is one of the most popular VR headsets with both all-in-one mode (compute onboard) and Oculus-link mode (compute on PC).

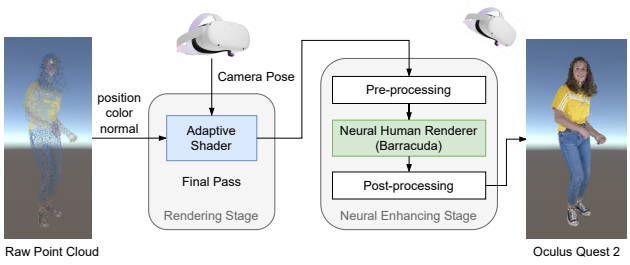


Figure 4. Point-based neural human rendering pipeline

3.3.1 Unity and Shader

Unity is one of the most widely used game engines for 3D gaming, architecture, and engineering. We use the XR Interaction Toolkit [2] to support the communication with our VR device, which provides a high-level, component-based interaction system. Processing the input information from the VR device, we can obtain the VR camera view points, local motions, and controller events easily through XR Interaction Toolkit and having the view-dependent rendering.

Shaders are a powerful tool in Unity that gains its efficiency through the parallel execution on GPU. Besides its original usage of calculating the color of each pixel rendered based on the lighting input and the material configuration, we can also encode the customized information and operation into the shader and perform the rendering via precomputed features. It is also possible to perform the simple network forward pass through Shader and RenderTexture.

3.3.2 Barracuda

We use the Barracuda package [37], a lightweight cross-platform neural network inference library for Unity, to perform neural enhancement as a post-processing step in our VR application. Given a pretrained neural network in ONNX format, Barracuda can perform neural network inference on both the CPU and GPU platforms. To run neural enhancement, we first store the current rendered output from Unity into a RenderTexture object and crop the original input from 1600×1600 to 720×720. The cropping operation not only eliminates the distortion effect at the margin but also helps speed up the inference processing. The cropped texture image is passed through the Barracuda worker, enhanced by our pretrained neural network from sparse point projection to photorealistic images, and pasted back to the output RenderTexture object.

3.3.3 Pipeline

The whole pipeline for our point-based neural human rendering in VR is shown in Figure 4. Given the point cloud, we pass the information of point position, color, and optionally normal into Unity. Our adaptive shader takes the per-point features, and the real-time camera poses from the VR device and computes the distance between each point to the camera center, which will be used to adjust the size and orientation of the rendered point adaptively. In the neural enhancing stage, we first load the pretrained network from the ONNX file into Barracuda and process the adaptive shader's output with the cropping, downsampling, and color mapping pre-processing. After neural model inference, the result is post-processed accordingly and pasted back to the Unity renderer's output, which will be displayed in the VR device.

Besides the human-specific rendering, our pipeline can be easily extended to the general point cloud scene with other neural enhancing effects, e.g., style transfer.

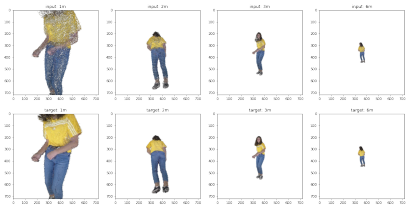
3.4. Implementation

We develop our application based on Unity 3D and build it for both Windows (run on PC) and Android (run on VR) platforms. DirectX11 and Vulkan graphics API are used on two platforms, respectively. When running on a PC, Oculus-link is used to communicate between the Windows app and Oculus device. We also design a user-interface for choosing different point clouds, loading the pretrained network, and manually adjusting the point size. Our adaptive shader can run at around 120FPS on a laptop with Nvidia GTX 1660Ti and 40FPS on native Oculus. With the neural rendering, we can achieve about 17FPS and 2FPS on a laptop and Oculus.

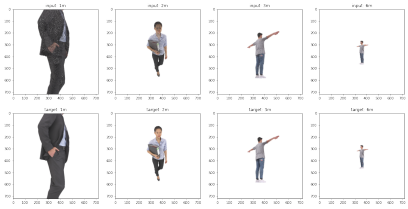
4. Experiments

4.1. Datasets

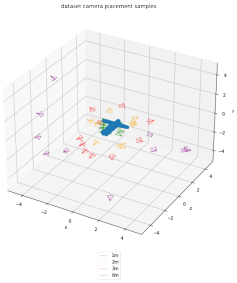
We use two sets of human source models to generate our datasets. First, we use a dense point cloud with 1M points obtained from 3D scans of a human. For input data, we render 10% of all points randomly. For output data, we render the full point cloud. Additionally, we use human models from RenderPeople.com [1], which provides high-resolution human meshes obtained from 3D scans, including dynamic human meshes. For input, we sample 100k points from the mesh surface using weighted random sampling based on the mesh triangle area and render the point cloud. For output, we render the full mesh data. We use two static human mesh data (denoted as "Dennis" and "Mei"), as well as one dynamic mesh sequence (denoted as "Dance"). To address color consistency with respect to depth, we capture our data from 4 different camera distances to the object: 1m, 2m, 3m, and 6m. For each depth value, we render 500 images from random camera elevation and azimuth angles.



(a) High quality point cloud data



(b) RenderPeople Human data



(c) Example camera placement

Figure 5. Example images of our dataset at different depths. notice that our adaptive shader can adjust point size with respect to camera depth

4.2. Adaptive Shader

In this section runtime performance and visual effects of the adaptive shader is dicussed.

4.2.1 Frame rate comparison

During speed test, different objects are examined in VR and the corresponding refresh rates are recorded. The test is run on desktop Nvidia GTX 1660Ti in Unity. To understand program speed with different visual effects, vanilla pcx shader, adaptive shader, adaptive shader with directional light, and neural rendering with standard layer (introduced in section 4.3 are tested. Results are shown in Table 2.

Outputs show that neural rendering realized by Baracuda [37] is a huge bottleneck for run-time speed in VR rendering. Comparatively, each shader runs fluently, even with a stress test of 30 dynamic and static point clouds combined.

Scene	static	dynamic	30 mixed
Vanilla Pcx	120	120	83
Adaptive shader	120	120	60
Neural rendering	17	15	7.1

Table 2. Frame rate comparison of different shader and neural rendering. Adaptive shader has the automatic point size, and Neural rendering runs on the output of adaptive shader.

Figure 6. Qualitative results of different shader

4.2.2 Visual comparison

For the ease of visualisation, We test our shader for four human point clouds in a different Unity project [9]. In Figure 6 the baseline result is the raw PLY binary little-endian format file rendered by Pcx. The left four columns are screenshots of same objects with closer camera-to-object distance as the right four columns. The second row shows our shader output without lighting information and Blinn-Phong shading. The third row shows our shader with some preliminary view-dependent shading information. In Blinn-Phong shading, vertex normal is important in deciding the shading color. Since the vertex normals of point cloud Humbi(the first and fifth column) are calculated from the nearest neighbors and are not stable, noticeable artifacts in this point cloud are present. Overall, our adaptive shader can render human point cloud well both close and far.

4.3. Neural Rendering

In this section will discuss some quantitative and qualitative results of our model that led to our choice of parameter and loss function. The baseline we compare our results to are the input images from our dataset, the metrics are always computed in regard to the ground truth of the corresponding images in the dataset.

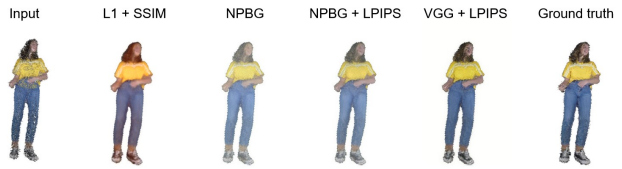


Figure 7. Qualitative results of different loss functions used while training.

Figure 8. Qualitative results of our method.

Figure 9. Screenshots captured directly from the virtual reality device. A) shows the rendering of only one person in the scene. B) shows a scene containing several different persons. C) Shows the result of a person model not contained in the training set.

4.3.1 Quantitative Results

For comparison of different loss functions in the model we use following metrics: L1, SSIM [45], LPIPS [48], VGG [30], NPBG [4]. To get a better comparison, we trained different models using a combination of the mentioned metrics directly as losses. The results are shown in Table 3. Our chosen loss function has the best evaluation with the LPIPS and VGG metric. From the results we see that a combination of VGG and LPIPS complement each other, whereas that is not necessary the case for LPIPS and NPBG.

4.3.2 Qualitative Results

Qualitative result of different loss functions are shown in Figure 7. From our evaluations, a model based on an LPIPS loss preserved the most details and had very saturated colors, whereas results based on the NPBG loss looked a bit pale. Simpler factors such as L1 and SSIM seem to favor features like color, but the results were often oversaturated. From the discussion in [48] and our own findings, we conclude that the LPIPS metric is the decisive factor for our model.

Outputs of our validation sets showed that the combination of VGG and LPIPS comes the closest to the ground thruth. The quantitative and qualitative evaluations seem to agree that a combination of LPIPS and VGG yields the perceptually best results, as shown in Table 3 and Figure 7.

The output of the complete pipeline can be seen in Figure 8 and Figure 9. We not only show that our pipeline can process and render one person from a sparse point cloud, but the scene can also contain several dynamic point clouds. We tested it with up to 30 persons in the same scene, and it worked as intended, see Figure 9 B).

Figure 10. Input, output and ground truth of our architecture where some layers have been removed.

4.3.3 Ablation Study

Data augmentations played a big part with our initial dataset when we used only constant depth to the object. Without augmentations, we found that the trained model was not robust to the different depth changes when walking around in the virtual space. However, with the new dataset and more training data, we found that augmentations became irrelevant for our test cases.

Shallower Architecture. To make our model compacter we tried to remove the last down convolution and the corresponding up convolution, see Figure 3 the blue block with 32 output channels and the green block with 16 output channels. It still yielded good results, as seen in Figure 10. However, this does not make any real difference on the performance since the main bottleneck is at the first convolution layer. Removing other parts of the network architecture did not yield usable results.

Generalization. Our model was only trained on static scenes, i.e. different persons kept the same pose. When trying it out on the VR device, the model did not have any problems processing dynamic scenes containing several persons as seen in Figure 9 B). Even when it faced a

Losses	$L1 \downarrow$	$1 - SSIM \downarrow$	$LPIPS \downarrow$	$VGG \downarrow$	$NPBG \downarrow$
Baseline	0.0112	0.0728	0.0656	0.0073	0.6697
L1 + SSIM	0.0152	**0.0412**	0.0308	0.0059	0.7592
L1 + SSIM + VGG	**0.0108**	0.0450	0.0520	0.0062	0.7005
VGG	0.0132	0.0488	0.0409	0.0057	0.6043
LPIPS	0.0137	0.0537	0.0342	0.0069	0.7670
LPIPS + NPBG	0.0214	0.0556	0.0502	0.0064	0.6388
LPIPS + VGG	0.0121	0.0459	**0.0277**	**0.0057**	0.6367
LPIPS + VGG + NPBG	0.0156	0.0514	0.0397	0.0060	**0.6003**
ALL	0.0156	0.0538	0.0519	0.0066	0.6443

Table 3. Metric evaluation of different loss functions. All models are trained for 100 epochs on the full training dataset. The metric scores are averaged over 120 unseen image pairs from the dataset.

completely new person it managed to produce good results as seen in Figure 8, second from left, and Figure 9 C).

4.4. Failure Cases

We observed some bleeding effects. During the generation of the input image, dark pixels from the cloth or hair, that are occluded in the real world, shined through. The model used this information and made the rendered skin and clothes darker than they actually are.

5. Conclusion

In this project, we develop a point-based neural human rendering pipeline in a real-time VR setting. Two enhancement modules, our adaptive point cloud shader and neural rendering module through Barracuda, are introduced to achieve the photorealistic rendered results from sparse 3D point cloud input. During the experiment, we conducted extensive studies on our adaptive shader's performance, different losses, and data augmentation for neural rendering. In addition, we test the efficiency of our pipeline for building on both the laptop platform and the native Oculus platform.

In the current data generation pipeline, there still exists a small inconsistency between the training data (from Pytorch3D) and the actual renderer input (from Unity). One potential future work is to improve the rendering technique and point cloud generation pipeline for training data to prevent the problem of bleeding and transparency. Regarding the model aspect, our current model is trained on a small dataset with limited network capacity. For the next step, we can explore better hardware-friendly architectures with larger capacity and train on the large-scale dataset of human-specific and general point cloud scenes.

Another interesting direction worth exploring is leveraging the shader's parallelism in Unity for efficient computation. In addition to the automatic point size rendering, the shader can support more complex operations such as the forward pass of an MLP network. Hence, it is possible to move the computation of neural rendering from the post-processing stage to the rendering stage. By precomputing the per-point (neural) features and defining the network inference operation accordingly in the custom shader, one may achieve a much faster neural rendering speed than the current Barracuda implementation.

6. Contributions of team members

- Long: Reconstruct VGG19 loss function from the NPBG paper; shader programming: geometry shader, polynomial fitting, and preliminary view dependent effect; Point cloud animation; Tests and demo generation.
- Weirong: Unity pipeline; training neural rendering model; testing and time profiling on laptop; helping with demo generation.
- Tianfu: Mesh and point cloud geometry processing; data generation pipeline on PyTorch 3D, PyTorch 3D adaptive point cloud shader; building on native Oculus hardware
- Markus: Neural Rendering model; training the model, trying out different loss functions and parameters, and finding out what works best; Ablation Study

7. Acknowledgement

We thank ETH Computer Vision and Learning Group for providing the materials and hardware for this project. We also specially thank our supervisor, Dr. Sergey Prokudin for proposing this interesting topic and the kind guidance through out the project.

References

[1] Renderpeople. https://renderpeople.com/3d-people. Accessed: 2022-06-10. 5

[2] Xr interaction toolkit: Xr interaction toolkit: 0.9.4-preview. 5

[3] Varun Agrawal, Soumith Chintala, and Abhishek Kadian. https://github.com/pytorch/examples/blob/main/fast_neural_style/neural_style/transformer_net.py, 2022. 2, 4

[4] Kara-Ali Aliev, Artem Sevastopolsky, Maria Kolos, Dmitry Ulyanov, and Victor Lempitsky. Neural point-based graphics. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXII 16*, pages 696–712. Springer, 2020. 1, 2, 4, 7

[5] James F. Blinn. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 11(2):192–198, jul 1977. 3

[6] Pere Brunet and Frederik W Jansen. *Photorealistic rendering in computer graphics: proceedings of the Second Eurographics Workshop on Rendering*. Springer Science & Business Media, 2012. 2

[7] Qifeng Chen and Vladlen Koltun. Photographic image synthesis with cascaded refinement networks. In *Proceedings of the IEEE international conference on computer vision*, pages 1511–1520, 2017. 2

[8] Tian Qi Chen and Mark Schmidt. Fast patch-based style transfer of arbitrary style. *arXiv preprint arXiv:1612.04337*, 2016. 2

[9] Long Cheng. Adaptive shader for human point cloud. https://github.com/loOong-Cheng/Adaptive-Shader-for-Point-Cloud-ply-/tree/master. Accessed: 2022-06-11. 6

[10] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009. 4

[11] Leon A Gatys, Alexander S Ecker, Matthias Bethge, Aaron Hertzmann, and Eli Shechtman. Controlling perceptual factors in neural style transfer. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3985–3993, 2017. 2

[12] Tom Geller. Overcoming the uncanny valley. *IEEE computer graphics and applications*, 28(4):11–17, 2008. 1

[13] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014. 1

[14] Brian Guenter, Mark Finch, Steven Drucker, Desney Tan, and John Snyder. Foveated 3d graphics. *ACM Trans. Graph.*, 31(6), nov 2012. 2

[15] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for mobilenetv3, 2019. 2

[16] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. *CVPR*, 2017. 1, 2

[17] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual losses for real-time style transfer and super-resolution, 2016. 4

[18] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. In *European conference on computer vision*, pages 694–711. Springer, 2016. 4

[19] Anton S. Kaplanyan, Anton Sochenov, Thomas Leimkühler, Mikhail Okunev, Todd Goodall, and Gizem Rufo. Deepfovea: Neural reconstruction for foveated rendering and video compression using learned statistics of natural videos. *ACM Trans. Graph.*, 38(6), nov 2019. 2

[20] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014. 4

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. 4

[22] Marc Levoy and Turner Whitted. *The use of points as a display primitive*. Citeseer, 1985. 2

[23] Yijun Li, Chen Fang, Jimei Yang, Zhaowen Wang, Xin Lu, and Ming-Hsuan Yang. Universal style transfer via feature transforms. *arXiv preprint arXiv:1705.08086*, 2017. 2

[24] Ricardo Martin-Brualla, Rohit Pandey, Shuoran Yang, Pavel Pidlypenskyi, Jonathan Taylor, Julien Valentin, Sameh Khamis, Philip Davidson, Anastasia Tkach, Peter Lincoln, et al. Lookingood: Enhancing performance capture with real-time neural re-rendering. *arXiv preprint arXiv:1811.05029*, 2018. 1

[25] Oscar Meruvia-Pastor. Enhancing 3d capture with multiple depth camera systems: A state-of-the-art report. In *RGB-D Image Analysis and Processing*, pages 145–166. Springer, 2019. 1

[26] Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. Mitsuba 2: A retargetable forward and inverse renderer. *ACM Transactions on Graphics (TOG)*, 38(6):1–17, 2019. 1

[27] Alexander Overvoorde. Introduction, vulkan tutorial. https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Introduction. Accessed: 2022-06-10. 3

[28] Hanspeter Pfister, Matthias Zwicker, Jeroen Van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 335–342, 2000. 2

[29] Francesco Pittaluga, Sanjeev J Koppal, Sing Bing Kang, and Sudipta N Sinha. Revealing scenes by inverting structure from motion reconstructions. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 145–154, 2019. 2

[30] Sergey Prokudin, Michael J Black, and Javier Romero. Smplpix: Neural avatars from 3d human models. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 1810–1819, 2021. 1, 4, 7

[31] Nikhila Ravi, Jeremy Reizenstein, David Novotny, Taylor Gordon, Wan-Yen Lo, Justin Johnson, and Georgia Gkioxari. Accelerating 3d deep learning with pytorch3d. *arXiv:2007.08501*, 2020. 3

[32] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015. 4

[33] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. pages 4510–4520, 06 2018. 2

[34] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014. 4

[35] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. 2019. 2

[36] Unity Technologies. Exporting your model to onnx format — barracuda — 3.0.0, 2022. 4

[37] Unity Technologies. Supported onnx operators — barracuda — 3.0.0_2022, 2022. 2, 4, 5, 6

[38] Ayush Tewari, Ohad Fried, Justus Thies, Vincent Sitzmann, Stephen Lombardi, Kalyan Sunkavalli, Ricardo Martin-Brualla, Tomas Simon, Jason Saragih, Matthias Nießner, et al. State of the art on neural rendering. In *Computer Graphics Forum*, volume 39, pages 701–727. Wiley Online Library, 2020. 1

[39] Ayush Tewari, Justus Thies, Ben Mildenhall, Pratul Srinivasan, Edgar Tretschk, Yifan Wang, Christoph Lassner, Vincent Sitzmann, Ricardo Martin-Brualla, Stephen Lombardi, Tomas Simon, Christian Theobalt, Matthias Niessner, Jonathan T. Barron, Gordon Wetzstein, Michael Zollhoefer, and Vladislav Golyanik. Advances in neural rendering, 2021. 1

[40] Dmitry Ulyanov, Vadim Lebedev, Andrea Vedaldi, and Victor S Lempitsky. Texture networks: Feed-forward synthesis of textures and stylized images. In *ICML*, volume 1, page 4, 2016. 2

[41] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance normalization: The missing ingredient for fast stylization. *arXiv preprint arXiv:1607.08022*, 2016. 2

[42] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance normalization: The missing ingredient for fast stylization, 2016. 4

[43] Ting-Chun Wang, Ming-Yu Liu, Jun-Yan Zhu, Andrew Tao, Jan Kautz, and Bryan Catanzaro. High-resolution image synthesis and semantic manipulation with conditional gans. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018. 1

[44] Ting-Chun Wang, Ming-Yu Liu, Jun-Yan Zhu, Andrew Tao, Jan Kautz, and Bryan Catanzaro. High-resolution image synthesis and semantic manipulation with conditional gans. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8798–8807, 2018. 2

[45] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004. 7

[46] Tim Weyrich, Jason Lawrence, Hendrik PA Lensch, Szymon Rusinkiewicz, and Todd Zickler. *Principles of appearance acquisition and representation*. Now Publishers Inc, 2009. 1

[47] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search, 2018. 2

[48] Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 586–595, 2018. 4, 7

[49] Matthias Zwicker, Hanspeter Pfister, Jeroen Van Baar, and Markus Gross. Surface splatting. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 371–378, 2001. 2