Review of ADOP: Approximate Differentiable One-Pixel Point Rendering

Balthazar Neveu - ENS Paris-Saclay
balthazar.neveu@ens-paris-saclay.fr

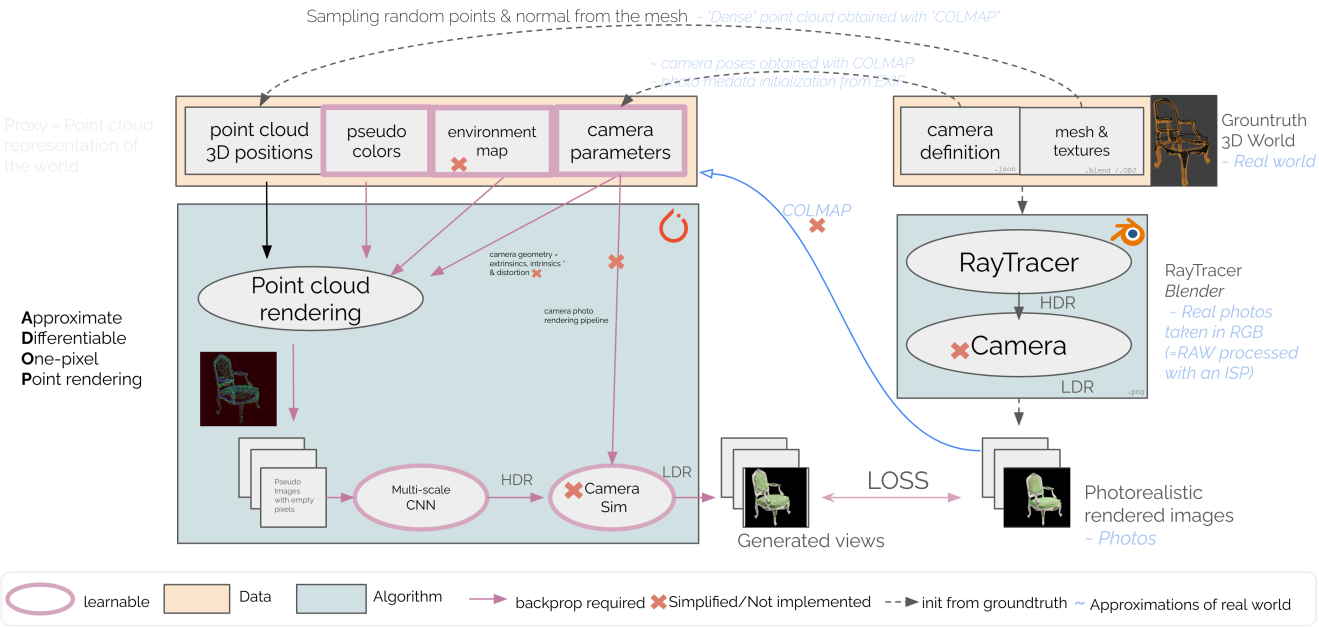

Figure 1: Overview of our partial re-implementation to study the ADOP [18] paper, in ideal conditions with calibrated scenes. *Left*: Point based neural rendering reconstructs novel views from a point cloud. Original paper implementation in ADOP works on real photos of large scale scenes. It therefore tries to model camera exposure and non linear tone mapping to adapt to each camera rendering.
***Right*: We mimick real world photo captures by rendering calibrated scenes and cameras (to place ourselves in the most simple conditions to review point based rendering). Point cloud is sampled randomly from the mesh with perfect normals and perfect camera poses (these would usually be estimated using COLMAP [19]). We do not let exposure vary or perform tone mapping.**

KEYWORDS

Neural rendering, Differentiable rendering, Point-based rendering

1 INTRODUCTION

The purpose of this report is a review of the paper ADOP: Approximate Differentiable One-Pixel Point Rendering by Rückert et al. [18]. Novel view synthesis is an intense topic of research since Neural Radiance Fields (NERF [15]) showed that a neural network could model a complex radiance field and lead to impressive novel view synthesis using volumetric rendering. NERF jointly recovers geometry and object appearace without any prior knowledge on the geometry. Other families of methods use a 3D geometric "proxy" of the scene such as a point cloud (NPBG [1]) or even meshes (NDS [22]). ADOP chooses point clouds but point based rendering leads to images filled with holes and at first sight does not really look like an appropriate data structure to render continuous surfaces of objects. However, we'll see how ADOP:

- manages to use a point cloud structure jointly with a CNN (processing in the image space) to sample dense novel views of large scenes from real photographs.
- makes a special effort to try to model realistic camera pipeline to improve the quality of the rendered images.
- does not inherently have an ability to model view dependent effects such as specularities or reflections.

A re-implementation from scratch in Pytorch of some of the key elements of the paper has been made in order to understand the core aspects of the ADOP paper (which were already present in a previous paper named Neural Point Based Graphics [1]). To simplify the study, it seemed like a good idea to work on **calibrated synthetic scenes**. This way, I have been able to focus on trying to evaluate the relevance of point based rendering, see their limitations and avoid the difficulties inherent to working with real world scenes (massive amount of data, large and noisy point clouds).
Finally, my code is fully available on GitHub and offers the possibility to generate novel views interactively.

2 CONTEXT

ECCV 2020 : The NERF [15] paper triggers an increasing interest in the field of novel views synthesis. Can you render novel viewpoints from a given calibrated scene using a neural network and a data driven approach? Can it look as good as running a complex rendering engine (such as raytracing)? The answer is yes and... and it also generalize on real scenes where there's no knowledge of the underlying scene. Many works try to improve the quality/speed of the rendering. We'll see how, suprisingly, point clouds can be used to render novel views of a scene.

2.1 What is novel view synthesis?

2.1.1 Real scenes. Novel view synthesis is a standard computer vision task which consists in generating new viewpoints of a scene after capturing a set of images. The concept is simple, take photographs or a video of a scene by walking around (or flying a drone).
Camera pose estimation. The first step is to get the camera parameters which are not necessarily known:

- extrinsics: pose for each capture [1]
- camera intrinsics : focal length, principal point[2], distortion

External initialization. Some of these variables can be pre-estimated, for instance intrinsic camera parameters can usually be pre-calibrated using the Zhang "checkerboard" method [24] and are assumed to be constant for the whole sequence of images [3].
An IMU (inertial measurement unit) can be attached to the camera which allows later to have an estimation of the camera pose after running a sensor fusion algorithm.
But there will always be measurement errors (sensor noise, calibration error, sensor fusion errors) [4]. So there's a need for an algorithm to estimate the camera poses from images, regardless of having an external pose estimation initialization.
Structure from motion. The traditional approach consists in using the popular COLMAP software [19] to jointly estimate camera trajectory and sparse point cloud. A side product of running this algorithm is getting a dense colored 3D point cloud of the scene [5]. The second step is to reconstruct a flexible representation of the scene so it can be rendered from new viewpoints. One of the technical challenge is to find the most suited data structure to represent the 3D scene subject to constraints such as:

- image / 3D structure quality: for cultural heritage or industrial monitoring applications for instance.
- reconstruction time and memory consumption. Real time for AR/VR applications is a constraint. *For instance, ADOP seems to take advantage of point cloud rendering hardware acceleration available in any computer using OpenGL (not necessarily with the need of a massive NVidia GPU).*
- preprocessing resources: in case users want to recreate their own scenes, they may not have access to powerful GPU

2.1.2 Calibrated scenes. Camera poses and scenes are perfectly known and controlled by using 3D scene synthesis. This is an easier setup to study novel view synthesis as you can truly evaluate the rendering quality of the algorithm without doubts on the quality of pose estimation (or camera imperfections). This setup is sometimes refered as "calibrated scenes".[6]

2.2 Representations of the scene

If we solely use the colored point cloud, we'll end up with images filled with holes (when we zoom in for instance). This is where rendering a scene start to get difficult.
Surface reconstruction. It is possible to get continuous representations (without holes) by wrapping a surface around the point cloud of an object. For instance assuming we have pre-computed normals, a Signed Distance Function (SDF) defined from point cloud can be evaluated anywhere. The surface of the object is where the SDF is equal to zero. Evaluating IMLS (Implicit Moving Least Square [12]) on a fixed grid followed by the marching cube algorithm allows to very simply recreate a mesh.[7].

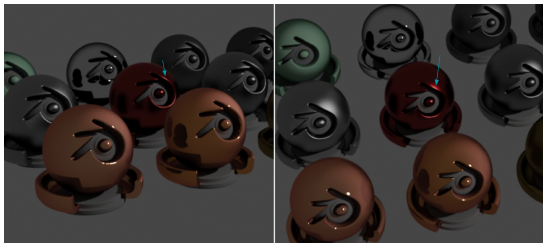

Figure 2: Colors change with camera orientation, specular materials reflect the light and amplify this effect, the extreme use-case being mirrors.

Meshes. Creating a mesh from the point cloud will lead to a nice geometric representation which can be rendered using classic rasterization techniques. But shading these triangles is still needed. If all materials are perfect diffusers, applying the textures extracted from the photos to the triangles shall be enough. Unfortunately, this will not work for specular materials as illustrated in figure 2. Neural deferred shading has been proposed [22] to jointly fit a mesh while optimizing the pixel shader (mimicked by a neural network) of a classic mesh rendering pipeline (geometry processing → rasterization → *(neural)* shader). Although mesh representations sounds flexible, the shading is baked into the scene representation and cannot be changed afterwards (for instance, lighting or materials cannot be changed).
Neural radiance fields. NERF [15] represents the scene RGB colors and density as a function of the 3D position and viewing angle. By shooting rays at the scene and sampling along them, one can integrate the estimated colors to render the final image[8]. A MLP

[1] 6 degrees of freedom = 3 rotations and 3 translations
[2] projection of the lens optical center onto the sensor which is not always located perfectly at the center of the image sensor
[3] Fixed calibration is not possible when the camera has a zoom-in capabilities or auto-focus. Variable focus may also affect the focal length.
[4] Please note that efforts can be made, such as calibrating camera/IMU alignment [8]
[5] multi-view stereo being the second part of the COLMAP software
[6] One could argue that representing scenes with sophisticated neural rendering is useless when you have the underlying 3D model and Blender available. Nevertheless, novel view synthesis on calibrated scenes is a good framework to test a method before deploying on real scenes.
[7] Limitations: All topologies may not be represented correctly (e.g. a hole in the cheese may end up being filled) and meshes can appear very smooth. More advanced alternative: Poisson Surface reconstuction [9]
[8] In this volumetric rendering, a density=0 means that the space is empty

(Multi Layer Perceptron) is used to represent the radiance field. Having the dependance on the viewing angles allows to model complex materials and lighting effects. Main drawback is that NERF are computationally expensive as they require evaluating a MLP all along the rays for each pixel...

Point clouds. Although point clouds are not continuous, they are a versatile representation of the scene and kind of easy to manipulate. The introduction of an additional inpainting method (e.g. neural rendering component) in the image space allows to overcome the limitations of the sparseness (see section 3) while being able to render fast.

2.3 Fast point based Rendering

One of the major advantage of point based methods is that this technology is backed by years of prior industy work. Rendering point clouds can be hardware accelerated and is available to the mainstream public on basically any computer (without NVidia GPU). Libraries such as OpenGL offer the option to render points `GL_Points` (and even specify the size of the splat `glPointSize(1);`)... Point clouds can even be rendered in a web browser as shown in figure 3.

Figure 3: Potree [21] allows point cloud rendering in the browser, points being rendered as tiny circles splatted on the screen.

3 ORIGINAL PAPER OVERVIEW

NPBG [1] (Neural Point-Based Graphics) is a novel view synthesis method based on point cloud representation and introduced several important concepts (1. and 3. below). ADOP [18] adds a few components (2. and 4.). to adapt to real scenes with high dynamic ranges and model complex camera pipelines.

1. Geometry. Learnable Pseudo-colors [9] assigned to each point are projected onto the camera screen at several scales.

2. Environment map. A learnable 360° image is used to model the scene background.

3. Neural rendering. The multi-scale pyramid of pseudo images will be jointly decoded and inpainted into a RGB HDR [10] image at full resolution, using a U-Net architecture.

4. Camera simulation. A camera simulation module will transform the image into a RGB LDR image [11].

[9]Pseudo-colors means a generic "feature" vector representation which can be more generic than three dimensional RGB components. A dimension 4 for instance.

[10]We define HDR as high dynamic range linear RGB images - as if they'd been retrieve from a RAW 12 or 14bit sensor with sole operations black point correction, demosaicing and potentially while balance/color matrix transforms.

[11]LDR stands for low dynamic range images, the ones we see on our screens after tone mapping, color adjustments like vibrancy, vigetting correction etc...

An overview of the pipeline is shown in fig. 1. The whole rendering pipeline is differentiable with regard to:

- the pseudo-color of each point
- the environment map colors.
- the photometry camera parameters (exposure, white balance correction, tone curve parametric vignetting...)
- the camera pose and intrinsics *This is an approximation, but it may be useful in order to refine camera pose estimation.*

Optimization Using several photos of the scene for supervision, all parameters (network weights, pseudo-colors, environment map and camera photo pipeline) are optimized to minimize a loss function (MSE $\mathcal{L}^2$ or perceptual loss[12]) between predicted LDR rendered images and the real ones.

Implementation. The authors wrote a C++ library (hybrid compiled code: lib Torch with cuda kernels and fast point rendering OpenGL code) to get fast trainings and reach impressive inference speeds: on scenes from Tanks and Temples [11] with 10 millions of points, they render the scenes at 37fps on a RTX3080 - 6 times faster than a NERF based method.

More details and limitations will be provided in the next sections.

4 REIMPLEMENTATION STRATEGY

My goal was to be able to train a minimalistic version of ADOP's differentiable point based neural renderer on calibrated scenes (to avoid the burden of large real scenes) [13].

Simplifications. Below is the list of all simplifications that have been made compared to the original ADOP paper.

- We assume linear RGB cameras with frozen exposure and without tone mapping.
- We discard environment map (e.g. our background is black).
- We generate photorealistic renders of synthetic meshes instead of using real photos.
- Camera poses are perfectly known.
- We do not refine the camera poses (no need to be approximately differentiable with regard to camera geometry parameters).
- Using meshes allows us sampling point clouds with normals without estimation errors such as the one we'd get with COLMAP.
- We can easily control the number of points to be able to test on a limited T500 GPU with 4Gb of RAM.

Implementation steps. [14]

- I used BlenderProc [4] to script and generate multiple synthetic scenes from the samples used in the NerF paper. All camera positions are known and share the same referential as my pytorch point renderer.
- A perfect point cloud is sampled at random from the mesh (through the .obj file).
- Several pytorch based function allow to project the points onto the image plane and include soft depth test and normal culling.
- A CNN is trained to predict the right dense color of the points in the image space.

[12]Perceptual loss [7] optimizes the distance between two images in a latent space rather than in the RGB colors space. We usually minimize the $\mathcal{L}^2$ distance between the feature maps in the middle of a frozen VGG network

[13]I decided to code everything from scratch knowing since the start that I would at no point be able to fully reproduce the results from the authors but I learnt a large amount of things by doing so.

[14]Each element has been carefully validated, mostly by visual tests or pytests.

4.1 Generating synthetic calibrated scenes

In order to avoid dealing with large real scenes with millions of points and having to deal with heavy COLMAP processing, I decided to use a recent library named BlenderProc [4] which wraps the Blender engine. [15]. The description of the data workflow is shown in appendix in fig. 17. It is first possible to generate a set of viewpoints configurations to orbit around the scene. It is also possible to provide an environment map (e.g. a skybox). Disabling the environment map as a background in Blender allows to remove one of the components of the ADOP paper. [16], as shown in fig. 4.

Figure 4: BlenderProc renders with an environment map.

We parameterize camera orientation using 3 Euler Angles (yaw pitch roll) and 3 positions to build the extrinsic camera matrix and use a pinhole camera model to project 3D points onto the image plane as we'll see in section 4.2. There's a perfect equivalence between the camera parameters used in Blender (fig. 5) and the ones used in my Pytorch renderer.

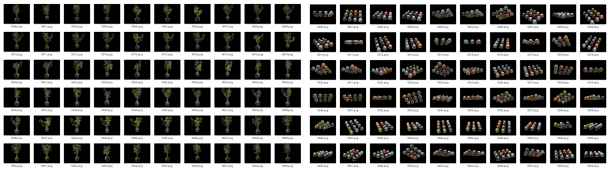

Figure 5: 60 views of resolution 640x480 of the `Ficus` scene rendered in 12 minutes. On the right, `material balls` scene rendered in a total of 25 minutes on a laptop equipped with a Nvidia T500.

One of the difficulties of using synthetic rendering from meshes is that sometimes thin surfaces are modeled with double sided triangles. In my implementation, points are sampled randomly from the mesh and associated with the normal of the single sided triangles to which the point belongs. Since our rendering pipeline uses normal culling, we end up with an issue. On the `ficus` scene for instance, "the green leaves" seen from the bottom will become invisible, so the renderer will face an impossible reconstruction task. We may therefore stick to objects with easy geometry such as the `old chair` scene. This is illustrated in fig. 6.
When working on natural photos, the points will most probably be placed on both sides of the surface by COLMAP with the normals in both ways.

[15] pip installed and launch without popping the graphical user interface
[16] It allows rendering the point cloud without using the neural environment map trick and we use black uniform color instead.

4.2 Projecting the point cloud onto the image plane

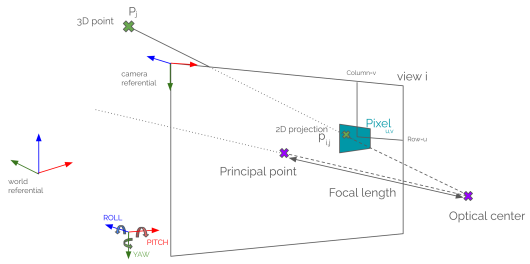

Figure 7: Pinhole camera model. The j^{th} point of the 3D point cloud, located at $\vec{P_{3D}}^{(j)}$ expressed in world coordinates is projected onto the sensor plane (i^{th} viewpoint) at position $\vec{p_{2D}}^{(i,j)}$ expressed as pixel coordinates, rounded to the 2D nearest integer coordinates (u, v)

4.2.1 Coordinates projection. To project a 3D point of index j located at $\vec{P_{3D}}^{(j)}$ in world coordinates onto the image sensor, we use the pinhole projection model as illustrated in 7.

$$\vec{p_{2D}}^{(j)} = K \cdot [Q_{\text{cam}}|T_{\text{cam}}]^i \vec{P_{3D}}^{(j)}$$

where

- K is the camera 3x3 intrinsic matrix.
- Q^i_{cam} is the camera orientation for view i, this matrix is created from the yaw, pitch, roll angles.
- T^i_{cam} is the 3D camera position for view i in the world frame (meters)
- $\vec{P_{3D}}^{(j)} \in \mathbb{R}^4$ are the homogeneous 3D coordinates of point j in the world frame.
- $\vec{p_{2D}}^{(i,j)} \in \mathbb{R}^3$ are the homogeneous 2D coordinates of point j projected in the sensor frame i (pixel coordinates).

This operation is performed in `forward_project.py` in parallel over all points j , using `torch.matmul` operation. [17]
Multiscale. Pseudocolored projected point cloud will be rendered at multiple scales. For a given scale $l \in \{0, 1, 2, 3\}$, $(u, v)^{(l)}$ are the rounded coordinates version of $\frac{\vec{p_{2D}}^{(i,j)}}{2^l}$.

4.2.2 Scatter operation. Visible points's colors are copied into corresponding pixels on the 2D image. A point only leads to a single pixel color update by taking the nearest pixel coordinate of $\lceil \vec{p_{2D}}^{(j)} \rfloor$. There are conditions to satisfy for each point to be valid.

- $(C1)$ The point must be inside the image frame $0 \leq x < W$ and $0 \leq y < H$.
- $(C2)$ The point must be in front of the camera ($Z > 0$).
- $(C3)$ The normal of the point must face the camera - see fig. 8.
- $(C4)$ The point must not be occluded by another point.

[17] Points coordinates could probably be projected all at once for all camera views using `torch.bmm` or `torch.einsum`.

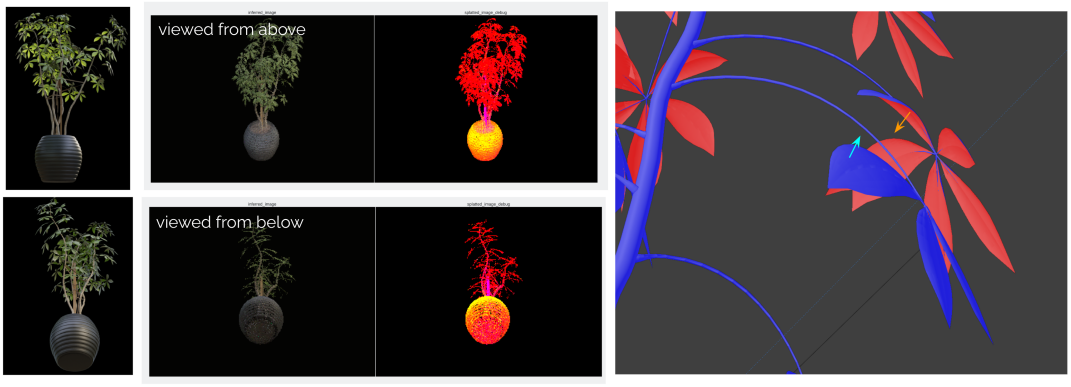

Figure 6: On the left, image from the Blender render of the ficus scene, seen slightly from below where the leaves look brighter than when looked from above. On the right side, we check the rendering of the point cloud with the "Bypass" mode (which allows ajusting the colors points to the scene). Most leaves are oriented upwards so normal culling does not render these points when seen from below. The optimization process has trouble to reconstruct correct colors for the leaves. On the right side, the blender face orientation (blue front facing the camera/red back facing the camera) fully reveals that the leaves of the `ficus` scene are made of double sided triangles.

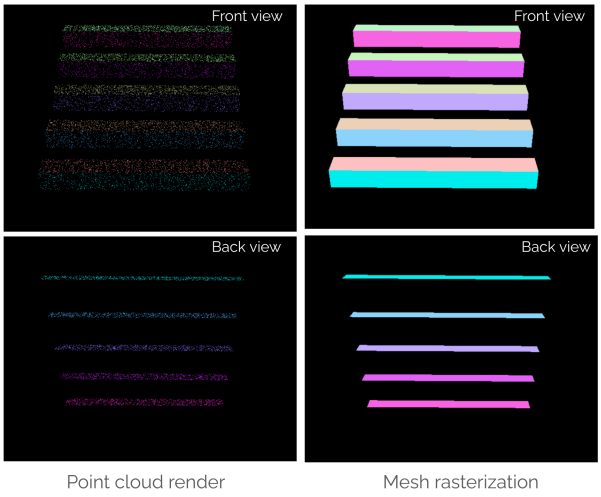

Figure 8: Validation of normal culling. We do not render points with normals pointing away from the camera.

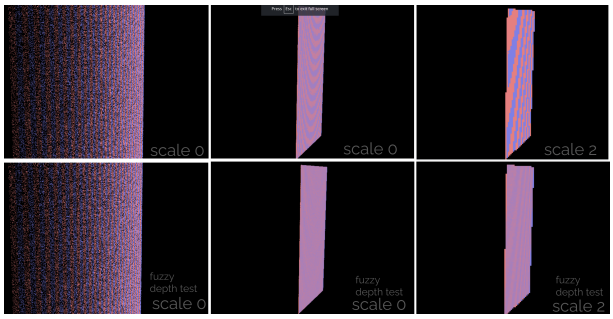

Figure 9: Fuzzy depth test acts as an anti-aliasing filter. On this test scene, a point cloud made of 500.000 point located in the same plane with an alternate vertical red an blue stripes on . We use $\alpha = 0$ on the top (hard depth test) and $\alpha = 0.01$ on the bottom (soft depth test). On the right side, when using larger scales (lower resolutions), aliasing effect if naturally amplified.

Soft depth test. That last condition $(C4)$ requires some work: Using a Z-buffer, it is possible to take the closest point to the camera. Since several points may fall into the same pixel cell, there will be aliasing (see fig. 9) as several pixels may be located on the same suface. The authors rely on previous work [20] to average point colors in a tiny range of depths located behind the closest point. This is called a soft depth test and we'll describe this part in details as it required a tricky implementation with Pytorch.

Soft depth test $Z \leq (1+\alpha) * \min_{Z_j}$ requires two passes. First compute the closest point to the camera for each pixel. Then, for each pixel, average the colors of the points that are close enough to the closest point. As we'll perform the first pass, we'll keep track of conditions $(C1)$, $(C2)$ and $(C3)$ to avoid recomputing them during the second pass.

First pass: Hard depth test.

`zbuffer_pass.py` is the first pass. It allows to find the closest point to the camera for each pixel. Depth values are initialized at ∞ and will remain ∞ when a pixel has not been filled. A first implementation trick is for each point of index j to keep a linear index k of its coordinate position in the image $k[j] = 1+u*W+v$ if the point is valid regarding conditions $(C1)$, $(C2)$ and $(C3)$ and set this index to $k[j] = 0$ otherwise. This allows to avoid recomputing the same test conditions during the second pass.

We then obtain the closest depth image for each pixel. $\forall (u,v) \in [0, H-1] \times [0, W-1]$:

$$Z^{\min}_{(u,v)} = min_{\{j \text{ s.t. } k[j]=1+u*W+v\}} \left(Z[j]\right)$$

Although this index mapping looks unnatural, it allows to use th native `torch.scatter_reduce_` operation to take the minimum of depth values for each pixel in a 1D tensor of size $1 + W * H$. [18]

Second pass. Soft depth test and color aggregation

In the second pass rendering/colors_aggregation.py, we are now capable to apply the soft depth test as we have already computed the closest distance of the point cloud to the camera for each pixel. We'll modify the point index mapping $k[i]$ to only keep indices which satisfy the soft depth test ($C4$):

$$k'[j] = k[j] * \left(Z[j] \leq (1+\alpha) * Z^{\min}_{(u,v) \equiv k[j]}\right)$$

Multiplying by the boolean tensor gracefully sets to 0 the indices which do not satisfy the condition and keeps the others unchanged. Keep in mind that the $k = 0$ index is the index for invalid pixels which will be discarded. Finally, we average the colors of the points that satisfy the soft depth test, at each pixel location, using `torch.scatter_reduce_` operator with the `reduce='mean'`. It is possible to keep a background

This rendering pipeline is fully differentiable with regard to the point cloud colors. We define the **"bypass" mode** as the following minimalistic training process:

- initialize the point cloud pseudo colors with random values.
- at each scale, compute a linear combination (conv1x1) of the pseudo colors to get the RGB rendered colors.
- the bias allows to fit a uniform background color.
- minimize the MSE loss between the rendered image and the target image at all scales using the ADAM optimizer with a learning rate of 0.01 for 100 epochs using a 55 views for training and 5 views for validation.

This simple sanity check revealed some issues: I couldn't properly render the `ficus` scene due to the double sided triangles issue mentioned previously. The `material balls` scene cannot be properly fit with the reflective materials as shown in fig. 12. On the `old chair` scene we obtain a validation PSNR of 16.7dB and relevant colors when visualizing reconstructions (see fig. 11). If we replace the 1x1 convolution by a 5x5 convolution to allow the points to have a larger spread than 1 pixel, we immediately see an improvement up to 21.2dB. Adding more samples to the point cloud obviously improves PSNR.

[18] My first `legacy_splatting.py` implementation with a for loop was way too slow and using native torch operators allowed me to get fast point cloud rendering (and save some development time to avoid rewriting a custom CUDA kernel).

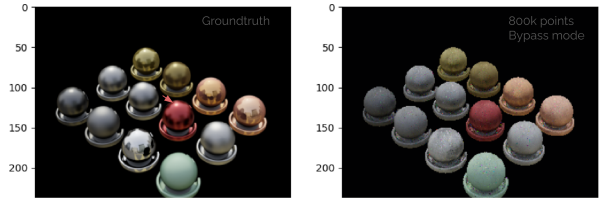

Figure 12: View dependant appearance: If we take a careful look at the `material balls` scene, no matter how much point we add, we cannot recover the view dependancy effect with the "bypass mode". It looks like we simply retrieve the average diffuse component.

It is not trivial to find a scene with lighting and materials which do not exhibit view dependant appearance, especially it seems like the sample scenes used in the NERF paper were carefully chosen to test the inherent ability of the neural renderer to learn something similar to a BRDF (which models in computer graphics the law of light reflecion for each material, including mirrors)..

4.3 Neural pixel rendering.

4.3.1 UNET.. The last operation is to render the final pixel colors by decoding the pseudo color images. ADOP [18] and NBPG [1] both use a modified UNET [17] which is a widely used multi-scale image processing architecture. Two modifications are made to the UNET architecture:

- Convolution operator is a gated convolution proposed in a 2019 NVidia paper dedicated to image inpainting [23]. This operator requires knowing the mask (e.g. which pixels are valid).
- At each scale, the UNET encoder takes pseudo-colored renders of the point cloud. Note that, at low resolution (higher levels of the pyramid), many points end up being averaged in the same pixel as described in the soft depth test. We end up with thumbnails with not much holes which will intuitively simplify rendering (this is not equivalent to downsampling the full resolution images filled with holes).

4.3.2 Vanilla networks. I started implementing a Vanilla architecture shown in fig. 13 to make sure I could improve the point cloud rendering right. Due to time constraints, I did not have enough time to implement and train the UNET described in the previous section section 4.3.

We report quantitative performances (PSNR on validation set) of various training configurations on the `Old chair` scene in table 1. By using the CNN rendering on a point cloud of 100.000 points, we're able to "fill the holes" and achieve a better PSNR than the bypass mode with 800.000 points. Qualitively, fig. 11, shows blurry results when using the Vanilla architecture despite better PSNR. More work is needed to improve the results and implement the UNET architecture with the right parameters.

Please note that all trainings were performed on a laptop with a Nvidia T500 GPU with only 4Gb of RAM. We decrease the batch size to 8 images when increasing the size of the architecture.

4.3.3 Results visualization. It is possible to render novel views interactively and live. Please refer to fig. 16.

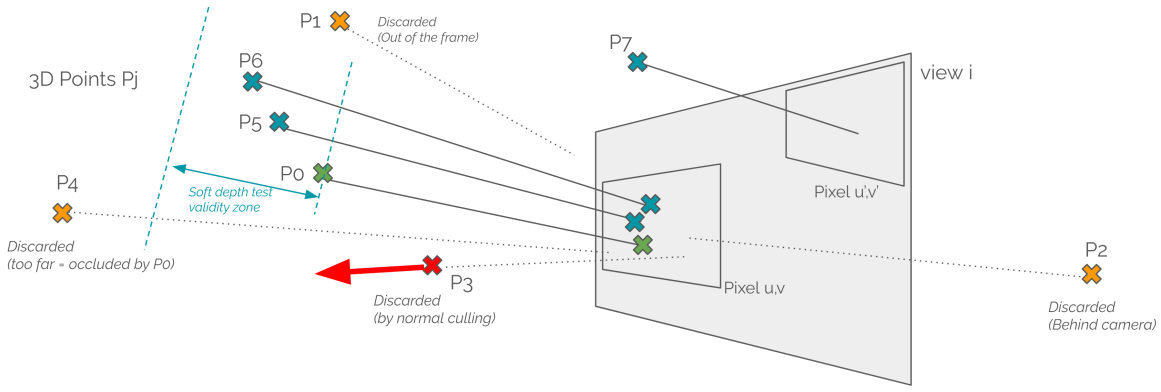

Figure 10: Soft depth test illustration: $P^{(0)}$ is the closest point to the camera for pixel (u, v). We will average the colors of the points $(P^{(0)}, P^{(5)}, P^{(6)})$. During the first pass, we'll keep track of the mapping between point indices and projected coordinates (in a 1D fashion). $k[0] = k[4] = k[5] = k[6] = 1 + u * W + v$ and $k[1] = k[2] = k[3] = 0$, $k[7] = 1 + u' * W + v'$. First pass will find the minimum of depths for each pixel $Z^{\texttt{min}}_{(u,v)} = min_{\{j \text{ s.t. } k[j]=1+u*W+v\}} (Z[j]), \forall (u, v) \in [0, H-1] \times [0, W-1]$. For instance $Z^{\texttt{min}}_{(u,v)} = Z^{(0)}$ and $Z^{\texttt{min}}_{(u',v')} = Z^{(7)}$.
During the second pass, we'll only keep indices which satisfy the soft depth test: $k'[j] = k[j] * \left(Z[j] \leq (1+\alpha) * Z^{\texttt{min}}_{(u,v)\equiv k[j]}\right)$. For instance, point 4 is considered too far away and discared e.g. $k'[4] = 0$.
Finally, colors are averaged $I_{u,v} \propto \sum_j \texttt{s.t } k'[j]=1+u.W+v\, I(j)$. For instance $I_{(u,v)} = \frac{I(0)+I(5)+I(6)}{3}$ and $I_{(u',v')} = I(7)$.

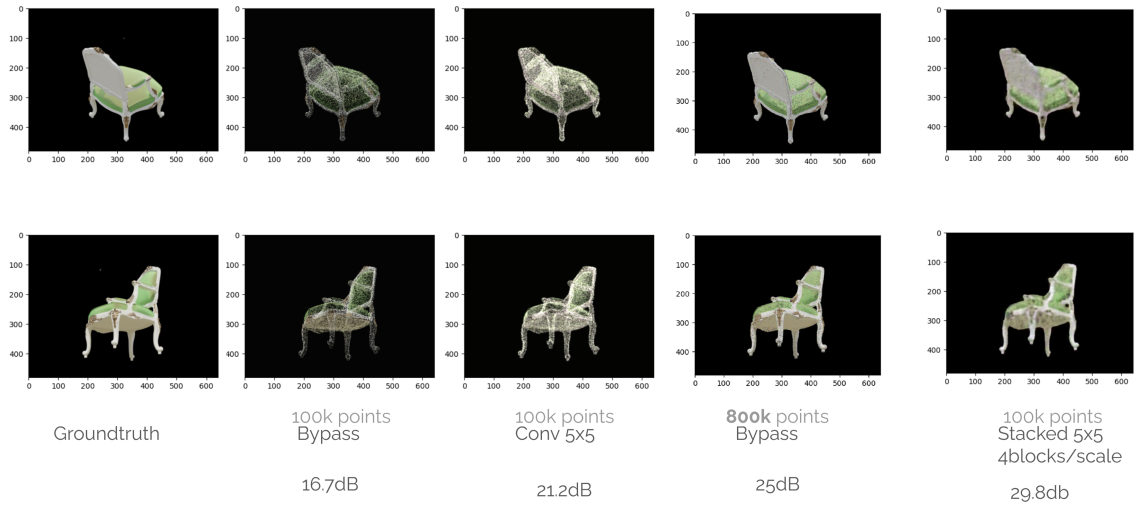

Figure 11: Two validation novel views using various training options (including the "bypass" mode and vanilla multiscale decoders (at each scale, we're stacking several stages of Conv5x5+Relu)).

5 DISCUSSION ON THE ORIGINAL PAPER

ADOP[18] is overall an excellent paper. It does not bring so much novel ideas but makes a considerable engineering effort to apply the core idea of NPBG [1] to large real scenes. In section 5.2, I will discuss some improvement ideas I had while working on the re-implementation of the ADOP paper. But we'll discuss a point of criticism in the next section.

5.1: Real scenes ≈ biased evaluation. Evaluation on real scenes from the Tanks and Temple dataset [11] mixes two things:

- the inherent neural rendering method (point based rendering) quality assesment.
- all improvements made by taking into account the camera pipeline.

Despite ablation studies to see the effect taking the camera photo pipline into account, the comparison to other methods is unfair and I'll propose a few ideas for a new benchmark to evaluate neural rendering quality in a more controlled environment for research purpose.

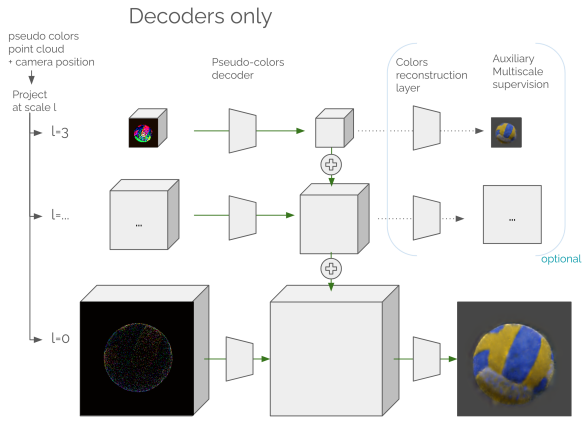

Figure 13: Vanilla architecture: Multiscale CNN with decoders only. Pseudo colors have more than 3 channels (4 or 8). At the top of the pyramid ($l = 3$), the projected point clould pseudo colors tensor is decoded to a tensor with 8 channels (still not a RGB thumbnail but a feature map). The lower scale $l = 2$ will be processed by another decoder - but in a residual fashion (meaning that the decoded feature vector from scale $l = 3$ will be upsampled (bilinear interpolation) and added to the feature vector of the decoder at scale $l = 2$). Finally at scale $l = 0$, the high resolution decoded feature map is processed by another Conv+Relu layers to end up with 3 channels. On the right side: an auxiliary decoder at each scale allows to add a potential multiscale supervision loss (not mentioned in the original ADOP paper)

5.1 Limitation: Benchmarking only on real scenes

Focusing only on real scenes. Suprisingly, the authors do not mention any attempts to apply their method to synthetic scenes, like the NERF paper initially did and they mention that training a new scene requires a large amount of compute. This could mean they developped their method progressively on a few toy examples and didn't bother releasing results of these tiny examples. Or their method may simply not perform too well on synthetic scenes... which may include a lot of specular materials [19]. I still think it would be beneficial to have a few synthetic scenes to test on including to make quick tests without the need of A100 40Gb trained for several hours. On the other hand, since the main contribution are refining pose estimation (requiring the approximate differentiable aspect of the renderer), handling large scenes and their camera simulator, it's not surprising that the authors showcase their work on natural photos rather than simulation.

Camera pipeline module. The attempt of the authors to model the camera pipeline comes from a good intention. By including the camera pipeline as part of their rendering, they are able to:

- work in a linear color space where... performing additions or convolutions is physically grounded.
- compensate exposure and colors shifts per scene.
- get a better accuracy on the Tanks and Temple benchmark which has a lot of exposure changes.

I think the authors picked the low hanging fruits here: take care of most exposure changes to end up improving the metrics on the available benchmark: Table 4. from their paper shows that ADOP is better by a large margin than all other algorithms (except the M60 tanks scene which has been shot in manual frozen exposure). They didn't backport their idea to the comcurrent methods they compare to so I don't think the comparison is really fair (the point based method may simply not be the key element leading to good results).

Anyway, their claim to handle the camera imaging pipeline properly is legitimate, extremely well crafted and justified in the paper. But it has a major caveat: results may crumble in case of a more complex or different camera ISP [20] than the one from Sony A7SII or DJI X5R. Please refer to fig. 14 to compare a standard ISP pipeline against the ADOP rendering.

The idea is interesting but it will most probably fail with a modern smartphone camera which use sophisticated local correction. Other kind of photo finish effects such as vibrancy make some colors slightly more saturated (e.g. blue for skies but avoid saturating red too much for skin tones), tone mapping may act locally [2] and sharpening may be region dependent. The authors show that they're able to get a sky with cyan shift which looks like the camera picture (they fit the tone mapper correctly). Modern smartphone ISPs manufacturers have worked on this "cyan cast" problem (compress blue highlights instead of clipping when applying white balance) so **modeling these algorithms will become harder and harder as they get more and more sophisticated.**

Ablation study from the authors (table 3 and figure 6 of [18]) shows that ADOP results get better when including the camera module in the pipeline.

If we carefully look at the Tanks and Temples dataset [11], pictures are frames extracted from a mp4 video from 2 different high end cameras, sometimes using auto-exposure (see fig. 15). Although it's a good initiative for researchers to keep on using fixed established benchmarks, Tanks and Temples was initially designed for 3D reconstructions (e.g. evaluate performances of Structure from motion like COLMAP estimation compared to a groundtruth lidar captured point cloud) rather than novel view synthesis.

Name	Cam (D=DJI X5R, S=Sony A7SII)	Area (m^2)	Height (m)	τ (mm)	Frames	Points (M)	ISO	f	Shutter (sec.)
Playground	D	54	2.8	10	7,463	1.7	200	f/2.8	Auto
Train	S	35	5.6	5	12,630	21.7	Auto	f/5.6	1/1000

Figure 15: Information on the scenes captured for the Tanks and Temples dataset with high end video cameras in addition to a Lidar point cloud. On the left, the "train" scene shows clear signs of overexposure. The playground scene on the right has an overall correct and steady exposure.

[19] Classical NERF test samples scenes usually contain a lot of specular materials where the rendering would most probably fail

[20] Image Signal Processor

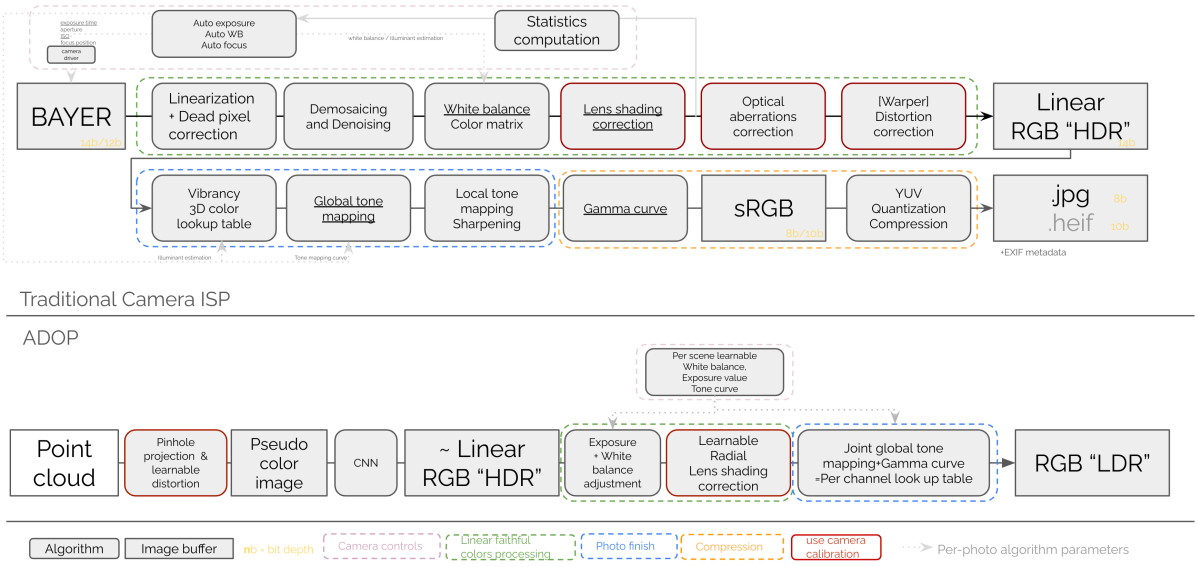

Figure 14: Comparing a modern ISP camera pipeline at the top versus the ADOP pipeline at the bottom which includes some learnable parameters such as the tone mapping curve, exposure compensation or white balance adjustments.

My recommendation is that all evaluations better be led in the linear domain. It is the best way to make fair comparisons between novel view synthesis algorithms. Benchmarking would strongly benefit from a dataset "upgrade": a sort of *"Linear Tanks and Temples"* carefully crafted by taking RAW shots with a high end DSLR. This is proposed next.

Switching to RAW format? One of the potential way of creating a new benchmark would be to capture the scenes with a DSLR (like a full frame sensor) shot both in RAW and jpg (usually available on most cameras). RAW files would be post-processed by Adobe LightRoom or DxO Photolab with a neutral rendering: no tone mapping and neutral color rendering to get a Linear RGB set of images without any image compression. In case the sensor 12 or 14bits dynamic range is not sufficient, HDR captures could be achieved by bracketing. Intuitively, it sounds natural to use a tripod and merge the LDR linear images into a HDR linear image. These HDR images may serve as the ground truth for validation. But even handheld, all raw captures with bracketing / at various exposures could be used (let the alignment of bracketing images be implicitly done by the 3D rendering engine, without any explicit need to merge exposures).

The main caveat of using RAW images is that there'll always be noise present (proper to the camera sensor) in the raw files. Since you get multiple views of the same scene, we can use the key concept proposed in Noise2Noise [13] that a neural network can be trained to denoise images on noisy images directly (without clean ground truth images - it requires noisy burst of the same image instead). Here we have access to the same scene under different angles. The engine to implicity align them and get the right supervision is the rendering of the point cloud itself. The idea of using NERF to work on RAW data has been proposed in "NERFs in the dark" [14].

RGB linear format is the only space in which an evaluation of the rendering quality does not depend on the camera ISP. Supervision could be linear RGB (denoised and demosaicked RAW files)... or simply RAW files (see the pioneer idea in Mosaic2Mosaic [5]).

5.2 Improvement ideas

A lot of my implementation work has been based on trying to apply the concepts of the ADOP paper to calibrated scenes. I think that it is possible to make these photorealistic images look more like the ones we'd get from real cameras.

Calibrated scenes + realistic simulated camera pipeline. . Regarding my simplified implementation, I did not try to include the camera pipeline module not only because of limited time but also because simulating a realistic camera pipeline is a very difficult task. Starting from calibrated scenes, here are the steps to mimick a realistic camera pipeline:

- output HDR linear images out of Blender [21].
- mimick RAW files: inverse the "blue linear" blocks of fig. 14. The idea of reversing the ISP has been proposed in [3]: apply inverse white balance and color matrices from typical values, mosaick, add realistic poisson+gaussian noise depending on sensor characteristics and ISO value...
- Mimick the ISP which goes from 12/14bits linear RAW bayer data to a 8 bit jpg (which is a difficult task). One could simply use a high end software raw converter by re-introducing a DNG as input to Darktable (open source) or Adobe LightRoom. Another way is to use a deep neural network proxy of a blackbox ISP (see the work [6] which mimicks the Huawei P20 ISP)

[21] `BlenderProc` does not even offer EXR outputs at the moment

This is definitely **a large chunk of work** but it could bring a lot of value if one wants to try to make a commercial product for novel view synthesis which tries to adapt to all sorts of cameras.

Pseudo colors intialization. . Instead of initializing the pseudo colors of each point randomly, a trick could be to pre-train an auto-encoder on the reference images. For each point in the point cloud projected onto a given view, we have a target patch in the reference view. This patch could simply be encoded into a latent vector which could be aggreagated in the pseudo-color vector. Pushing the idea further, the pretrained decoder (from the auto-encoder) coud be used as the decoder in the neural rendering network. Here's the intuition: if the neural render encoder is the identity, this is equivalent to copy pasting the decoded patches at the right locations of projected points. Fine tuning the network will finish the job of stitching the patches together.

Inherent limitations to model view dependant material appearance. . By construction, the ADOP pipeline does not have a natural ability to model view dependent effects such as specularities or reflections. We have observed this in fig. 12. We could provide extra inputs to the neural network to model view dependent materials. By providing the angle between the point normal and the view direction (these 2 vectors are already computed during the normal culling test in the first hard depth test pass). These 2 angles could be concatenated to the projected pseudo colors vectors (and we'd probably need to map these to sine positional embeddings as proposed in NERF [15]). *I eventually found out about NPBG++ [16].* The authors of NPBG++ propose to fit the coefficients of spherical harmonics function which model color variation of each point with regard to relative camera operation.

Limitation when zooming too much. . The inherent limitation of point appears when you zoom in too close. A continuous parametric representation such as an anisotropic 2D Gaussian Kernel would allow mixing the advantage of point clouds (scale and speed) while trying to remove their discrete nature... Gaussian splatting [10] seems to be one of the most appropriate answer to this problem and removes the use of a CNN.

6 CONCLUSION

In conclusion, the review of this paper has been a great opportunity to dive into the world of neural rendering and understand the challenges of point based neural rendering.
I have implemented two major items:

- A script to render calibrated scenes which are ready for neural rendering (with perfect normals, perfect point cloud position and camera poses).
- A pytorch implementation of point projections with the fuzzy depth test (which was satisfying and fast enough to perform live inference)

I was therefore able to train a simple multiscale decoder CNN jointly with point pseudo colors on simple calibrated scene.
I have pointed out:

- a minor criticism regarding all tricks deployed to model the camera pipeline instead of truly questioning the relevance of benchmarks on the Tanks and Temple dataset.
- a limitation regarding handling view dependent effects which is not a problem in NERF.

The amount of work spent by the authors to make their method work very fast on large scale point clouds is tremendous. The craftsmanship of the computer graphics research community is very inspiring.

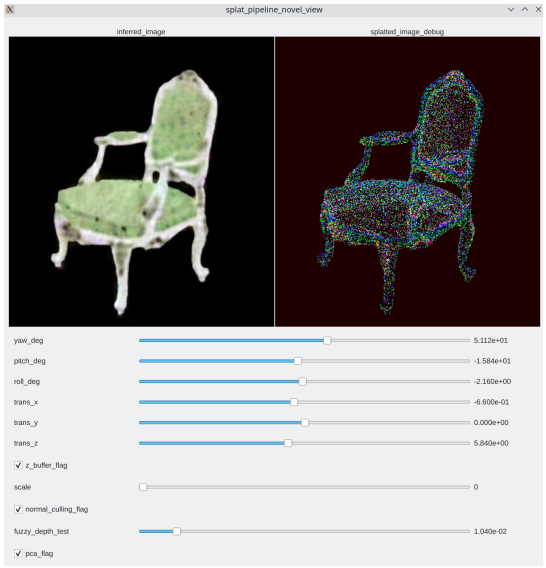

Figure 16: Live inference, the user can move the camera around the scene and see the novel view rendered in real time. On the right side we visualize the projected pseudo colored point cloud of dimension 8 using a Principal Component Analyzis.

REFERENCES

[1] Kara-Ali Aliev, Artem Sevastopolsky, Maria Kolos, Dmitry Ulyanov, and Victor Lempitsky. 2020. Neural Point-Based Graphics. (2020). arXiv:1906.08240v3 [cs.CV]

[2] Mathieu Aubry, Sylvain Paris, Samuel W Hasinoff, Jan Kautz, and Frédo Durand. 2014. Fast local laplacian filters: Theory and applications. *ACM Transactions on Graphics (TOG)* 33, 5 (2014), 1–14.

[3] Tim Brooks, Ben Mildenhall, Tianfan Xue, Jiawen Chen, Dillon Sharlet, and Jonathan T. Barron. 2018. Unprocessing Images for Learned Raw Denoising. arXiv:1811.11127 [cs.CV]

[4] Maximilian Denninger, Dominik Winkelbauer, Martin Sundermeyer, Wout Boerdijk, Markus Knauer, Klaus H. Strobl, Matthias Humt, and Rudolph Triebel. 2023. BlenderProc2: A Procedural Pipeline for Photorealistic Rendering. *Journal of Open Source Software* 8, 82 (2023), 4901. https://doi.org/10.21105/joss.04901

[5] Thibaud Ehret, Axel Davy, Pablo Arias, and Gabriele Facciolo. 2019. Joint Demosaicking and Denoising by Fine-Tuning of Bursts of Raw Images. arXiv:1905.05092 [cs.CV]

[6] Andrey Ignatov, Luc Van Gool, and Radu Timofte. 2020. Replacing Mobile Camera ISP with a Single Deep Learning Model. arXiv:2002.05509 [cs.CV]

[7] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. 2016. Perceptual Losses for Real-Time Style Transfer and Super-Resolution. arXiv:1603.08155 [cs.CV]

[8] Alexandre Karpenko, David Jacobs, Jongmin Baek, and Marc Levoy. 2011. Digital video stabilization and rolling shutter correction using gyroscopes. *CSTR* 1, 2 (2011), 13.

[9] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. 2006. Poisson surface reconstruction. In *Proceedings of the fourth Eurographics symposium on Geometry processing*, Vol. 7.

[10] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 2023. 3D Gaussian Splatting for Real-Time Radiance Field Rendering. https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/

[11] Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. 2017. Tanks and Temples: Benchmarking Large-Scale Scene Reconstruction. *ACM Transactions on Graphics* 36, 4 (2017).

[12] Ravikrishna Kolluri. 2008. Provably good moving least squares. *ACM Transactions on Algorithms (TALG)* 4, 2 (2008), 1–25.

[13] Jaakko Lehtinen, Jacob Munkberg, Jon Hasselgren, Samuli Laine, Tero Karras, Miika Aittala, and Timo Aila. 2018. Noise2Noise: Learning Image Restoration without Clean Data. arXiv:1803.04189 [cs.CV]

[14] Ben Mildenhall, Peter Hedman, Ricardo Martin-Brualla, Pratul P. Srinivasan, and Jonathan T. Barron. 2022. NeRF in the Dark: High Dynamic Range View Synthesis from Noisy Raw Images. *CVPR* (2022).

[15] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. 2020. NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis. (2020). arXiv:2003.08934 [cs.CV]

[16] Ruslan Rakhimov, Andrei-Timotei Ardelean, Victor Lempitsky, and Evgeny Burnaev. 2022. NPBG++: Accelerating Neural Point-Based Graphics. arXiv:2203.13318 [cs.CV]

[17] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-Net: Convolutional Networks for Biomedical Image Segmentation. arXiv:1505.04597 [cs.CV]

[18] Darius Rückert, Linus Franke, and Marc Stamminger. 2022. ADOP: Approximate Differentiable One-Pixel Point Rendering. (2022). arXiv:2110.06635 [cs.CV]

[19] Johannes Lutz Schönberger and Jan-Michael Frahm. 2016. Structure-from-Motion Revisited. In *Conference on Computer Vision and Pattern Recognition (CVPR)*.

[20] Markus Schütz, Bernhard Kerbl, and Michael Wimmer. 2021. Rendering Point Clouds with Compute Shaders and Vertex Order Optimization. arXiv:2104.07526 [cs.GR]

[21] Markus Schütz, Stefan Ohrhallinger, and Michael Wimmer. 2020. Fast Out-of-Core Octree Generation for Massive Point Clouds. *Computer Graphics Forum* 39, 7 (Nov. 2020), 13. https://doi.org/10.1111/cgf.14134

[22] Markus Worchel, Rodrigo Diaz, Weiwen Hu, Oliver Schreer, Ingo Feldmann, and Peter Eisert. 2022. Multi-View Mesh Reconstruction with Neural Deferred Shading. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[23] Jiahui Yu, Zhe Lin, Jimei Yang, Xiaohui Shen, Xin Lu, and Thomas Huang. 2019. Free-Form Image Inpainting with Gated Convolution. arXiv:1806.03589 [cs.CV]

[24] Zhengyou Zhang. 2000. A Flexible New Technique for Camera Calibration. *IEEE Trans. Pattern Anal. Mach. Intell.* 22, 11 (2000), 1330–1334. http://dblp.uni-trier.de/db/journals/pami/pami22.html#Zhang00

Number of points	PSNR	Mode	Conv size	Pseudo color dimension	Multi-scale supervision
100k	16.7dB	Bypass	1x1	3	Yes
100k	21.2dB	Conv 5x5	5x5	3	Yes
100k	26.3dB	Vanilla	5x5	8	No
100k	**29dB**	Vanilla	5x5	8	Yes
400k	23dB	Bypass	1x1	3	No
800k	25db	Bypass	1x1	3	No

Table 1: Quantitative performances (PSNR on validation set) of various training configurations on the `Old chair` scene.

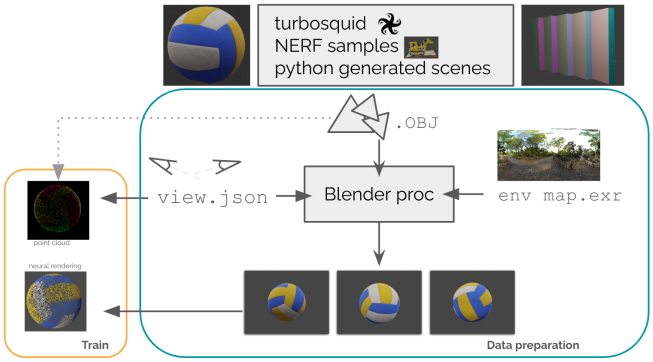

Figure 17: Workflow: `photorealistic_rendering.py` allows preparing multiple camera poses saved as `.json` files in order to render photorealistic views of a `.blend` or mesh `.obj` files which come from internet resources (TurboSquid/NERF sample dataset) or test scenes generated in python. The .obj and view .json files tie together the BlenderProc rendering and my Pytorch point rendering implementation: The point cloud is sampled from the mesh and the camera poses are known. A neural network is trained using `optimize_point_based_neural_renderer.py` to predict colors of the points by trying to match the multiple photorealistic renderings. CNN weights, pointcloud, normals and pseudo-colors are all saved along in a `.pt` file which later allows performing live novel view synthesis `novel_views_interactive.py` based on my own library interactive_pipe

.