In this paper we present ADOP, a novel point-based, differentiable neural rendering pipeline. Like other neural renderers, our system takes as input calibrated camera images and a proxy geometry of the scene, in our case a point cloud. To generate a novel view, the point cloud is rasterized with learned feature vectors as colors and a deep neural network fills the remaining holes and shades each output pixel. The rasterizer renders points as one-pixel splats, which makes it very fast and allows us to compute gradients with respect to all relevant input parameters efficiently. Furthermore, our pipeline contains a fully differentiable physically-based photometric camera model, including exposure, white balance, and a camera response function. Following the idea of inverse rendering, we use our renderer to refine its input in order to reduce inconsistencies and optimize the quality of its output. In particular, we can optimize structural parameters like the camera pose, lens distortions, point positions and features, and a neural environment map, but also photometric parameters like camera response function, vignetting, and per-image exposure and white balance. Because our pipeline includes photometric parameters, e.g., exposure and camera response function, our system can smoothly handle input images with varying exposure and white balance, and generates high-dynamic range output. We show that due to the improved input, we can achieve high render quality, also for difficult input, e.g., with imperfect camera calibrations, inaccurate proxy geometry, or varying exposure. As a result, a simpler and thus faster deep neural network is sufficient for reconstruction. In combination with the fast point rasterization, ADOP achieves real-time rendering rates even for models with well over 100M points.

https://github.com/darglein/ADOP

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2022 Association for Computing Machinery. 0730-0301/2022/7-ART99 $15.00 https://doi.org/10.1145/3528223.3530122

1 INTRODUCTION

With neural rendering it is possible to generate stunning freeviewpoint reconstructions of real world scenes. The field has advanced enormously in the last years with two major streams emerging [Tewari et al. 2020]: Proxy-based approaches that use some 3D proxy to carry reconstructed scene information (e.g., triangle meshes [Thies et al. 2019] or point clouds [Aliev et al. 2020]), and direct approaches that encode light or radiance fields (e.g., NeRFs [Mildenhall et al. 2020]).

Approaches with geometric proxies extend the idea of previous image based rendering methods [Chaurasia et al. 2013], with the difference that neural networks are used to interpolate and blend the input images on the geometric proxy. The rationale is that neural networks can achieve a better interpolation of view-dependent effects as well as handle imperfections of the geometric proxy and the resulting reprojection errors. Although this holds true to some degree, inconsistencies in the input still have significant impact on the quality of the result. As such, slight miscalibration of intrinsic and extrinsic camera parameters can lead to blurred results. Even worse, varying camera exposure in the input images cannot be compensated well and leads to patches of inconsistent brightness.

In this paper, we present a novel neural rendering system that directly accounts for such inconsistencies and renders very efficiently. As input we assume a number of images of a scene. A standard 3D reconstruction pipeline estimates intrinsic and extrinsic camera parameters and reconstructs a point cloud of the scene geometry. Following prior point-based neural rendering work [Aliev et al.
2 RELATED WORK

2.1 Novel View Synthesis

Traditional novel view synthesis, which is closely related to image-based rendering (IBR), relies on the basic principle of warping colors from one frame to another. One approach is to use a triangle-mesh proxy to directly warp the image colors to a novel view [Chaurasia et al. 2013; Debevec et al. 1998; Shum and Kang 2000]. However, imperfect geometry leads to ghosting artifacts around silhouettes. This can be improved by replacing hand-crafted heuristics in the classic IBR pipeline with deep neural networks [Hedman et al. 2018; Riegler and Koltun 2020, 2021; Thies et al. 2020]. If no proxy geometry is available, learning-based approaches have been developed that create a multi-plane image representation [Mildenhall et al. 2019; Srinivasan et al. 2019; Tucker and Snavely 2020; Zhou et al. 2018] or directly estimate the required warp-field [Flynn et al. 2016; Ganin et al. 2016; Zhou et al. 2016]. Novel view synthesis can also be performed by reconstructing a 3D model of the scene and rendering it from novel view points. Thies et al. [2019] learn a neural texture on a triangle mesh which can be rendered using traditional rasterization. The rasterized image is then converted to RGB by a deep neural network. Other approaches use ray-casting to automatically learn a voxel grid [Nguyen-Phuoc et al. 2018; Sitzmann et al. 2019; Zhu et al. 2018] or an implicit function [Mescheder et al. 2019; Park et al. 2019]. It has also been shown that point clouds are suitable geometric proxies for novel view synthesis [Mehry et al. 2019; Pittaluga et al. 2019]. Neural Point-based Graphics (NPBG) [Ali et al. 2020], which is closely related to our method, renders a point cloud with learned neural descriptors in multiple resolutions. These images are then used to reconstruct the final output image by a deep neural network. NPBG is very flexible as it doesn’t require a textured triangle mesh as proxy and shows impressive results. Similarly, Dai et al. [2020] show that rendering multiple depth layers of a point cloud can also be used to synthesize novel views.

2.2 Inverse Rendering

Inverse rendering and differentiable rendering have been a topic of research for some time. However, major breakthroughs have only been made in recent years due to improved hardware and advancements in deep learning [Kato et al. 2020]. Its application can be challenging though: Traditional triangle rasterization with depth-testing has no analytically correct spatial derivative [Kato et al. 2020; Loper and Black 2014]. Available systems therefore either approximate the gradient [Genova et al. 2018; Kato and Harada 2019; Kato et al. 2016; Loper and Black 2014] or approximate the rendering itself using alpha blending along edges [Chen et al. 2019; Liu et al. 2019a; Rhodin et al. 2015]. Volume raycasting on the other hand is differentiable by accumulating all voxel values along the ray [Kato et al. 2020]. This has been used by multiple authors to build volumetric reconstruction systems [Henzler et al. 2019; Lombardi et al. 2019; Tuinsiani et al. 2017] or predict the signed distance field of an object [Jiang et al. 2020]. Instead of a voxel grid, an implicit function can also be used inside a differentiable volumetric raycasting framework [Li et al. 2019b; 2020; Niemeyer et al. 2020; Zakharev et al. 2020]. Mildenhall et al. [2020] show with Neural Radiance Fields (NeRF) that a deep neural network can be trained by volumetric raycasting to store the view-dependent radiance of an object. Due to the impressive results of NeRF, multiple extensions and improvements have been published in the following years [Martin-Brualia et al. 2021; Sitzmann et al. 2020; Yu et al. 2021; Zhang et al. 2020]. Inverse rendering has also been proposed for point cloud rasterization [Kato et al. 2020; Lin et al. 2018]. The spatial gradients of the points can be approximated in different ways. This includes gradient computation along silhouettes [Han et al. 2020], gradients for Gaussian splatting [Kopanas et al. 2021; Wiles et al. 2020; Yifan et al. 2019], and gradient approximation using a temporary 3D volume [Insafutdinov and Dosovitskiy 2018]. A different approach is taken by Lassner and Zollhöfer [2021] who render the points as small spheres instead of splats. Our differentiable point rendering approach is similar to established differentiable splatting techniques [Kopanas et al. 2021; Yifan et al. 2019], however we render only one pixel per point, which allows our approach to be multiple magnitudes more efficient than competing methods [Lassner and Zollhöfer 2021]. A nice property of a differentiable rendering pipeline is that the camera parameters...
can be optimized during rendering. Lin et al. [2021] and Jeong et al. [2021] show that the camera model and camera pose can be optimized in the NeRF pipeline by providing a differentiable projection module. This allows to correct small errors in the initial calibration or even estimate the scene’s geometry from scratch. Similar work focuses on synthesizing photometric correct views by storing linear HDR radiance values inside the NeRF. A fixed tonemapper [Mildenhall et al. 2021] or neural tonemapper [Huang et al. 2021] then converts the integrated HDR intensities to the LDR color space. Concerning input refinement, this paper is inspired by these approaches and similar to concurrent work of Tancik et al. [2022]. However, we use an improved differentiable tonemapper module and apply the structural optimization on a point cloud instead of NeRFs.

2.3 Point-Based Rendering

Point-based rendering has been a topic of interest in computer graphics research for some time [Levoy and Whitted 1985]. Over the past decades two orthogonal approaches have been developed [Kobbelt and Botsch 2004]. The first major approach is the rendering of points as oriented discs, which are usually called splats or surfels [Zwicker et al. 2001], with the radius of each disc being precomputed from the point cloud density. To get rid of visible artifacts between neighboring splats, the discs are rendered with a Gaussian alpha-mask and then combined by a normalizing blend function [Alexa et al. 2004; Pfister et al. 2000; Zwicker et al. 2001]. Further extensions to surfel rendering exist, for example, Auto Splats, a technique to automatically compute oriented and sized splats [Preiner et al. 2012]. In recent years, deep learning-based approaches have been presented that improve image quality of surfel rendering, especially along sharp edges [Bui et al. 2018; Yang et al. 2020].

The second major approach to point-based graphics is point sample rendering [Grossman and Dally 1998], where points are rendered as one-pixel splats generating a sparse image of the scene. Using iterative [Rosenthal and Linsen 2008] or pyramid-based [Grossman and Dally 1998; Marroquín et al. 2007; Pintus et al. 2011] hole-filling approaches, the final image is reconstructed as a post processing step. To reduce aliasing in moving scenes, points with a similar depth value can be blended during rendering [Botsch et al. 2005; Schütz et al. 2021]. It has been shown that software implementations [Günther et al. 2013; Schütz et al. 2021] outperform hardware accelerated rendering through GL_POINTS [Shreiner et al. 2009]. Recently developed approaches replace traditional hole-filling techniques with deep neural networks to reduce bluriness and better complete large holes [Le et al. 2020; Meshry et al. 2019; Pittaluga et al. 2019; Song et al. 2020].

3 NEURAL RENDERING PIPELINE

An overview of our end-to-end trainable neural rendering pipeline is shown in Fig. 2. As input, we use a point cloud, which can come from common Multiview Stereo or LiDAR and SLAM systems, and the novel frame’s camera parameters. Learned neural descriptors are assigned to the point cloud.

The first step in the pipeline is a differentiable rasterizer (Fig. 2 left), that renders each point as a one-pixel-sized splat by projecting it to image space using the camera parameters. Second, a deep neural renderer is used to fill in holes and shade the image. Finally, a learnable tone mapping operator with a camera sensor model is applied, which converts the rendered high dynamic range (HDR) image to the displayable low dynamic range (LDR). For optimization, input views are re-rendered, compared to the ground truth, and the loss is backpropagated through our rendering pipeline. Since all steps are differentiable, we can simultaneously optimize all parameters, in particular the scene structure, photometric model, point features, and network parameters, which improves consistency of the input and adds robustness from a possible imperfect initial reconstruction or calibration. In the following section, we detail on these three steps.

3.1 Differentiable One-Pixel Point Rasterization

Forward Pass. Our differentiable rasterizer renders multiple resolutions of a textured point cloud using one-pixel-sized splats. Formally speaking, the resolution layer \( I \in \{0, 1, \ldots, L-1\} \) of the neural image \( I \) is the output of the render function \( \Phi_l \)

\[
I_l = \Phi_l(C, R, t, x, n, E, \tau),
\]

where \( C \) is the camera model, \((R, t)\) the camera pose, \( x \) the point position, \( n \) the point normal, \( E \) the environment map, and \( \tau \) the neural texture.
The rasterizer performs three steps, which are projection, occlusion check, and blending. The first step is the projection of each world point \( x \) into the image-space of layer \( l \). Using the camera model \( C \) and the rigid transformation from world to camera-space \((R, t)\), we define this projection as:

\[
P_l(C, R, t, x) = \frac{1}{z} C (Rx + t)
\]

(2)

The real valued result of \( P \) is then converted to pixel coordinates by rounding it to the nearest integer. A world point \( x_i \in \mathbb{R}^3 \) is therefore projected to pixel coordinates \( p_i \in \mathbb{Z}^2 \) by

\[
p_i = \left[ P_l(C, R, t, x_i) \right].
\]

(3)

Note that the rounding operation makes the projection discrete, which requires us to approximate its derivative (see later). After all points have been projected to image space, we discard them if they fail at least one of the following conditions:

- **Bounds Test:** \( p_{ix} \in [0, w[ \land p_{iy} \in [0, h[ \)

- **Normal Culling:** \( (Rn_i)^T \cdot \frac{Rx_i + t}{|Rx_i + t|} > 0 \)

- **Depth Test:** \( z \leq (1 + \alpha \min_z(p_i)) \)

(4)

(5)

(6)

The last condition (6) is the fuzzy depth-test as described in Schütz et al. [2021]. A point passes the fuzzy depth test if its \( z \)-value is smaller than or equal to the scaled minimum depth value at that pixel. If the camera model does not provide a valid \( z \)-value, we instead use the distance between the camera and the 3D point. A large value of \( \alpha \) in Eq. (6) increases the number of points that pass the depth test. This leads to a smooth image but can also introduce artifacts when background points are merged into the foreground. In practice we use \( \alpha = 0.01 \) as suggested by Schütz et al. [2021].

After the bounds check (4), normal culling (5), and fuzzy depth-test (6), for each pixel \((l, u, v)\) we obtain a list of points \( \Lambda_{l,u,v} \). If a pixel is not hit by any point, we sample the output color from the environment map \( E \) using the inverse camera projection \( C^{-1} \). Otherwise, we sample the texture \( \tau \) of every point and write the mean into the output image. The blending function \( B_l \) can therefore be written as:

\[
B_l(u, v, \Lambda) = \begin{cases} 
\frac{E(R^T C^{-1}((u, v)))}{|\Lambda_{l,u,v}|} \sum_{p \in \Lambda_{l,u,v}} \tau(p) & \Lambda_{l,u,v} \neq \emptyset \\
0 & \text{else}
\end{cases}
\]

(7)

Fig. 3 shows two color images that have been rendered using the one-pixel point rasterization technique described in this section. We can see that with \( \alpha = 0.01 \) (right image) the aliasing is significantly reduced and for example the letters are easier to read.

**Environment Map.** To represent the environment, we place a spherical environment map of resolution \( 1024^2 \) around the scene. Analog to the scene points, the environment map contains neural features, which are rendered as background. Both, the features of the point cloud and those of the environment map are learned using derivatives computed in the following backward pass.

**Backward Pass.** The backward pass of our point rasterizer first computes the partial derivatives of the render function (1) w.r.t. its parameters.

\[
\frac{\delta \Phi}{\delta C}, \frac{\delta \Phi}{\delta R}, \ldots
\]

Using the chain rule, we can then compute the parameter’s gradient w.r.t. the loss and pass it to the optimizer. The difficult part of this calculation are the partial derivatives of the structural parameters \( C, R, t \) and \( x \). This will be explained in the following section. The derivatives of the texture \( \tau \) and environment map \( E \) are straightforward and will not be detailed in this work.

The problem of deriving \( \Phi \) w.r.t. the structural parameters is the rounding operation Eq. (3) of our one-pixel point rasterizer. Due to the blending operation (7), it is not possible to compute the change of a pixel’s color by moving a point out of this pixel, without storing a list of points with each pixel and reevaluating the blending function. So instead, we use an approximation that works well in practice. The principle is shown in Fig. 4. We virtually shift the projected point at \( p = (u, v) \) one pixel in each direction and compute the change of intensity at the target pixels, which is possible if we know the depth value at the target pixel. The induced intensity change of these shifts are:

\[
\frac{\Delta I}{\Delta u} \bigg|_{p=(u+1,v)}, \frac{\Delta I}{\Delta u} \bigg|_{p=(u-1,v)}, \ldots
\]

(8)

from which we can approximate the gradient for \( u \) (equally for \( v \)) at the desired location with

\[
\frac{dI}{du} \bigg|_{p(u,v)} \approx \frac{1}{2} \left( \frac{\Delta I}{\Delta u} \bigg|_{p=(u+1,v)} + \frac{\Delta I}{\Delta u} \bigg|_{p=(u-1,v)} \right)
\]

(9)

(10)

To compute each induced change of (9), multiple cases have to be considered. As shown in Fig. 4, neighboring elements of \( p \) can either show a color from the environment map (background) or from other rasterized points. In the first case, the point simply overwrites the background, otherwise, three different cases have to be considered. Firstly, the virtually shifted point can be completely behind the neighbor in which case no intensity change would be induced. Secondly, the shifted point can be in front of the neighbor...
replacing the old color with the point’s texture. Lastly, the neighbor point can have a similar depth value according to the fuzzy depth test (6), which causes us to have to compute the change of the blend function (7) if \( p \) is added to the neighboring \( \Delta \). In summary, the four cases to compute local spatial gradients are (with \((i, j) = (u \pm 1, v \pm 1):\

\frac{\Delta f}{\Delta u}|_{p=(i,j)} = \begin{cases} 
\tau (u, v) - I_l (i, j) & \Lambda_{u,v} = 0 \\
0 & z > (1 + \alpha) \min_z (u, v) \\
\tau (u, v) - I_l (i, j) & z(1 + \alpha) < \min_z (u, v) \\
\frac{|\Lambda_{u,v}| I_l (i, j) + \tau (u, v)}{1 + |\Lambda_{u,v}|} - I_l (i, j) & \text{else}
\end{cases}
\]

(11)

Stochastic Point Discarding. Especially at small resolution layers, hundreds of points can pass the fuzzy depth-test of a single pixel, resulting in dramatic overdraw. To reduce this number, we apply a stochastic point-discarding technique similar to Enderton et al. [2010]. In a preprocess, we compute the world-space radius \( r_{\text{world}} \) as the distance to the 4th-nearest point. This radius is projected into screen-space during rendering, giving an approximation of the circular splat-size \( r_{\text{screen}} \). Based on this size we then discard points stochastically to obtain a desired number of points per pixel by assigning each a uniform random value \( \beta \in [0, 1] \) and rendering it only if the following condition is met:

\[
\frac{r_{\text{screen}}}{\sqrt{1 - \beta}} > \frac{1}{Y}
\]

(12)

The parameter \( Y \) roughly controls how many points are rendered per pixel. In our experiments, we use \( Y = 1.5 \) for all datasets and scenes, see also Fig. 9.

Implementation Details. To implement the forward pass efficiently, we mainly follow the work of Schütz et al. [2021]. They propose a three-pass point rasterizer that outperforms traditional point rendering with GL_POINTS. After a depth-only render pass, the color is accumulated for all points that pass the fuzzy depth-test. During accumulation, we track the number of points per pixel so we can compute Eq. (7) without writing the set \( \Lambda_{u,v} \) to memory. To further improve efficiency, we adopted the proposed blocked-morten-shuffle on the point cloud and use optimized local reductions inside the rasterization kernels. A side-effect of not explicitly storing \( \Lambda_{u,v} \) is that during backpropagation all points have to be rendered again. However, this re-rendering only marginally impacts performance as the total backpropagation time is dominated by the neural networks.

To further improve the pose estimation of our inverse rendering pipeline, we follow state-of-the-art SLAM systems [Engel et al. 2017; Mur-Artal and Tardós 2017; Rückert and Stamminger 2021] that optimize the rigid transformation \((R, t) \in SE(3)\) in tangent-space. Linearized pose increments are expressed as the Lie-algebra elements \( x \in se(3) \) and applied to a transformation by

\[
(R', t') = \exp (x) \cdot (R, t),
\]

(13)

where \( \exp (x) \) is the exponential map from \( se(3) \) to \( SE(3) \). To implement tangent-space optimization in libTorch [Jia et al. 2014][Paszke et al. 2019], we create a tangent tensor for the pose which is updated by the optimizer. After each step, the tangent is applied to the original pose (13) and reset to zero.

3.2 Neural Renderer

The neural renderer step (Fig. 2 center) uses the multi-resolution neural images to produce a single HDR output image. It consists of a four layer fully convolutional U-Net with skip-connections, where the lower resolution input images are concatenated to the intermediate feature tensors. Downsampling is performed using average-pooling and the images are upsampled by bilinear interpolation. As convolution primitives, we use gated convolutions [Yu et al. 2019] which have been initially developed for hole-filling tasks and are therefore well suited for sparse point input. Overall the network architecture is similar to Aliev et al. [2020] with one less layer (see Sec. 4.6) and a few modifications to enable HDR imaging. First, we remove the batch-normalization layers, as they normalize the mean and standard deviation of intermediate images to fixed values. The total sensor irradiance is therefore lost and cannot be propagated from a 3D point to the final image. Usually, we store the neural point descriptors linearly, but if the scene’s radiance range is sizable (larger than \( 1 : 400 \)) we use a logarithmic scaling, otherwise convergence issues with the optimizer can occur. For logarithmic descriptors, we convert them to linear space during rasterization so that the convolution operations only use linear brightness values.

3.3 Differentiable Tone Mapping

The HDR output image of the neural renderer is converted to LDR by a learnable tone mapping operator (Fig. 2 right), that mimics the physical lens and sensor properties of digital photo cameras. The first tone mapping stage applies brightness correction to the HDR image \( I_{\text{HDR}} \) using the estimated exposure value \( EV_L \) of image \( i \).

\[
I_e = \frac{I_{\text{HDR}}}{2^{EV_L}}
\]

(14)

If image meta information is available, we initialize \( EV_L \) using Eq. (15), where \( f \) is the \( f \)-number of the optical system, \( t \) the exposure time in seconds, \( S \) the ISO arithmetic speed rating, and \( EV \) the mean exposure value of all images. Otherwise, we initialize \( EV_L \) to zero for all images.

\[
EV_L = \log_2 \left( \frac{I_e^2}{I_t} \right) + \log_2 \left( \frac{S_t}{100} \right) - EV
\]

(15)

After brightness correction, we compensate for a changing white balance by estimating the white point \((R_w^w, G_w^w, B_w^w)\) for each image. During optimization, we keep \( G_w^w \) constant so that the white point cannot change the total image brightness.

As a next step, we model the vignette effect of digital cameras, which is a radial intensity falloff due to various optical and sensor-specific effects. The model we use [Goldman 2010] is a polynomial of the distance \( r \) of a pixel \( p \) to the vignette center \( c_v \):

\[
I_v = I_w \cdot \left( 1 + a_2r^2 + a_4r^4 + a_6r^6 \right)
\]

(16)

For better stability, we normalize \( p \) and \( c_v \) to be in the range \([0, 1]\). The coefficients \( a_i \) are initialized to zero and \( c_v \) is initialized to the image center.
The last stage in the tone mapping operator maps the linear RGB values to non-linear image intensities by applying the camera response function (CRF) [Grossberg and Nayar 2003] and an optional color space conversion from RGB to sRGB. We combine both operations into a single function $R$, which is implemented using a 1D texture for every color channel.

$$I_{ldr} = R(I_o)$$  \hspace{1cm} (17)

To guide the optimization towards a plausible response function, we initialize $R(x)$ to $x^{0.45}$ and add the following additional constraints based on Debevec and Malik [2008]: $R(0) = 0, R(1) = 1, R'(x) = 0$. These constraints ensure that the whole intensity range is covered and the response function is smooth. Overexposed and underexposed pixels are clamped to the maximum and minimum output values of 1 and 0. However, depending on the random network initialization, it is possible that the whole image is over/under exposed generating a zero gradient over the complete image. Inspired by the LeakyReLU activation function [Xu et al. 2015], we define a separate response function $R_t$ during training that leaks small values instead of clamping them.

$$R_t(x) = \begin{cases} 
\alpha x & x < 0 \\
R(x) & 0 \leq x \leq 1 \\
\frac{-\alpha}{\sqrt{x}} + a + 1 & 1 < x 
\end{cases}$$  \hspace{1cm} (18)

The last term of Eq. (18) asserts that the maximum leaked value is $(1+\alpha)$. This is important in HDR scenes, because an overexposure of multiple magnitudes should not create a large gradient. In practice we use $\alpha = 0.01$, which results in a maximum image intensity of $R_t(\infty) = 1.01$.

## 4 EXPERIMENTS

### 4.1 Datasets and Evaluation Methodology

We have collected seven datasets with differences in size, modality, and camera settings (see Tab. 1). The first four scenes, i.e. *M60*, *Train*, *Playground*, and *Lighthouse*, are from the popular tanks and temples dataset [Knapitsch et al. 2017]. They consist of around 310 images in Full-HD resolution captured by a high-end video camera. We use COLMAP [Schö nerberger and Frahm 2016] to reconstruct the initial dense point-cloud as well as the camera extrinsics and intrinsics. The three remaining scenes, *Kemenate*, *Boat*, and *Office*, are added to show how ADOP can handle different capture setups as well as to evaluate some specific aspects of our pipeline. The

| Scene | #Points | #Images | Resolution | Exposure | White Balance | Camera Model | Initial Reconstruction |
|-------|---------|---------|------------|----------|---------------|--------------|-----------------------|
| M60   | 9,713,277 | 313 | 2048 × 1080 | Constant | Constant | Pinhole + Dist. | COLMAP |
| Train | 11,818,812 | 301 | 1920 × 1080 | Auto     | Constant | Pinhole + Dist. | COLMAP |
| Playground | 12,521,941 | 307 | 1920 × 1080 | Auto     | Constant | Pinhole + Dist. | COLMAP |
| Lighthouse | 12,313,620 | 309 | 2048 × 1080 | Auto     | Constant | Pinhole + Dist. | COLMAP |
| Kemenate | 34,144,401 | 405 | 2880 × 1920 | Constant | Constant | Pinhole + Dist. | Metashape |
| Boat | 53,036,216 | 742 | 2880 × 1920 | Auto     | Constant | Pinhole + Dist. | COLMAP |
| Office | 72,916,173 | 688 | 5472 × 3648 | Auto     | Auto | Fisheye | LiDAR + SLAM |

Table 1. Overview of our evaluation scenes. *M60*, *Train*, *Playground*, and *Lighthouse* are from the Tanks and Temples dataset. *Kemenate* and *Boat* were captured by the authors. *Office* was provided by (anonymous).

Table 2. Timings for novel view synthesis averaged over the four tanks and temples scenes, and approximate training times. For SVS and Nerf++ we used the provided standard parameters for training.

|                      | Inference (RTX 3080) | Training (A100) |
|----------------------|----------------------|-----------------|
| Nerf++               | ~ 183,000 ms         | ~ 24 h          |
| SVS                  | ~ 2,400 ms           | > 48 h          |
| NPBG                 | 50 s                 | ~ 3 h           |
| ADOP (ours)          | 27 ms                | ~ 4 h           |

*Boat* scene was captured outdoors with a large variance in exposure time. The *Office* dataset consists of very high resolution fisheye input images and its point cloud was produced by a LiDAR scanner instead of a multi-view stereo approach. For *Kemenate*, the initial reconstruction was processed by Metashape [Agisoft 2021]. *Please see also the accompanying video for results of all scenes*. As shown in Table 1, several scenes were captures with varying exposure, though only in *Boat* the exact exposure value is stored in the meta data. In the other cases, we let ADOP estimate the real exposure value for each image. For a quantitative evaluation we follow the standard approach in our community. All input images are divided into a training set and a test set. The training set is used to optimize all parameters of our rendering pipeline over multiple epochs. We then let our system synthesize each test image and compare it to the real image using different metrics. Since the test images are also prone to miscalibration, we do a realignment and exposure estimation of the test images after training has been completed. For a fair comparison, we show the results with and without test refinement during our comparison to other approaches (see Tab. 4).

### 4.2 Training Details and Timings

Our inverse rendering pipeline (see Figure 2) is optimized end-to-end using the C++ front-end of libTorch. The neural network and the 4-element point descriptors are initialized randomly and optimized using ADAM [Kingma and Ba 2015] with an initial learning rate of 0.0002 and 0.08, respectively. If the radiance variation is high (see Sec. 4.11), the descriptors are stored logarithmically and their learning rate is reduced by a factor of 10. The remaining parameters, for example, camera pose and exposure time are updated using the SGD optimizer. Over time, each learning rate is reduced when a plateau is detected. For the exact values the reader is referred
Table 3. Quantitative ablation study on the major contributions of ADOP. The improvements due to the environment map (Env), the tone mapper (TM), and the structure optimization (SO) differs between the scenes. For example, the M60 scene was captured with fixed exposure hence the tone mapper only has a small impact. The structure optimization is beneficial in all cases. A qualitative evaluation for the Train scene is shown in Figure 6.

|            | Train VGG ↓ Diff | Playground VGG ↓ Diff | M60 VGG ↓ Diff | Lighthouse VGG ↓ Diff | Kemenate VGG ↓ Diff | Boat VGG ↓ Diff | Office VGG ↓ Diff |
|------------|------------------|-----------------------|----------------|----------------------|--------------------|----------------|------------------|
| Baseline   | 514.3            | 374.8                 | 369.5          | 508.5                | 204.7              | 652.9          | 186.8            |
| + Env.     | 490.8 -5%        | 374.8 0%              | 364.1 -2%      | 440.1 -14%           | 202.5 -1%          | 646.1 -1%      | 185.5 -1%        |
| + Env. + TM| 412.0 -19%       | 352.2 -6%             | 360.3 -3%      | 381.1 -25%           | 189.3 -8%          | 555.7 -15%     | 145.4 -22%       |
| + Env. + TM + SO| 368.1 -28% | 318.7 -15%           | 317.7 -14%     | 350.0 -31%           | 180.2 -12%         | 535.6 -18%     | 131.7 -30%       |

4.3 Loss Function

We have trained ADOP on the M60 scene using typical loss functions also used in prior work: L1, MSE, and the VGG perceptual loss [Johnson et al. 2016]. For evaluation we use the LPIPS loss [Zhang et al. 2018], the peak signal-to-noise ratio (PSNR), and the structural similarity index (SSIM). The results are shown in Fig. 5. Using the VGG loss, the most details are visible and the overall sharpness is significantly improved compared to the other losses. All further experiments thus use VGG as loss function.

4.4 Ablation Study

Compared to previous work of neural point-based rendering, we have proposed several ideas to improve the quality of novel view synthesis. In Tab. 3 and Fig. 6, we analyze how much each module contributes to the final result. The first row shows a baseline experiment, which is similar to NPBG [Aliev et al. 2020]. After that we enable one by one, the environment map (Env), the tone mapper (TM), and the structure optimization (SO). In the latter, we initialize the scene using the COLMAP reconstruction and then optimize the camera pose, camera model, and point cloud during the training stage. All together, a significant improvement in image quality is achieved with a VGG loss reduction between 12% and 30%. The impact of the individual modules differs between the scenes. The indoor scenes (M60, Kemenate, and Office) are not improved by the environment map. The tone mapper shows the largest impact on scenes with a high variance in exposure times (Train, Lighthouse, Boat, Office). However, structure optimization yields a positive change on all datasets.

4.5 Robust Pose Correction

To further prove the robustness of our approximate gradient computation (see Sec. 3.1), we add Gaussian noise to the camera position and rotation. As shown in Fig. 7, ADOP is robust to a bad initial
calibration and can also be used for image to point cloud alignment (see the supplemental material).

4.6 U-Net Layers
Due to the improved consistency of the input, we can afford to use a smaller and faster neural network for final reconstruction. While other methods often use a five layer network [Aliev et al. 2020; Thies et al. 2019], we show in Fig. 8 that also fewer layers can produce good results. Note that with the number of layers in particular the hole-filling capabilities decrease, so that the chosen number of layers should be selected according to the point densities. In our experiments, we found that a four layer network improves efficiency and is less prone to overfitting, all shown results thus were generated with four layers.

4.7 Stochastic Point Discarding
In Sec. 3.1 we proposed a stochastic point discarding technique to reduce the pixel overhead. To examine a possible quality loss, we train ADOF with and without point discarding on the Kemenate dataset. We measure the number of blending operations, render time, and image quality, the result is presented in Fig. 9. Visually, no difference is noticeable, which is backed by the perceptual rendering loss. However, on average only 50% of the points are blended, which reduces rendering time by 20%.

4.8 Comparison With Prior Work
We evaluate our system with state-of-the-art prior work on the scenes from the tanks and temples dataset [Knapitsch et al. 2017]. see Tab. 1. The methods we compare against are SynSin [Wiles et al. 2020], Pulsar [Lassner and Zollhöfer 2021], Neural Rendering in the Wild (NRW) [Meshry et al. 2019], NeRF++ [Zhang et al. 2020], Stable View Synthesis (SVS) [Riegler and Koltun 2021], and Neural Point Based Graphics (NPBG) [Aliev et al. 2020]. The first two are general differentiable rendering front-ends that have been adapted by us to the novel view synthesis problem. They can be seen as similar to our approach and NPBG with the difference of using sphere and splat-based rendering instead of multi-resolution one-pixel point rendering. We did not compare against all related inverse rendering systems, because some are not applicable to our data [Loper and Black 2014] and others are outperformed by Pulsar and Synsin [Kato et al. 2020; Lin et al. 2018; Liu et al. 2019a].

Pulsar, Synsin, Nerf++ and SVS only support pinhole cameras. Therefore we train them on the undistorted images and camera parameters provided by COLMAP. During evaluation the synthesized undistorted images are distorted again and compared to the ground truth. NPBG, NRW, and our approach use one-pixel point rendering.
Fig. 10. Comparative results of novel-view synthesis on the \textit{Train}, \textit{Playground}, and \textit{M60} scene. The table below provides a quantitative comparison, in this figure we show images for the methods that perform best. Note that due to memory constraints SVS has been evaluated in half-resolution.

Table 4. Quantitative evaluation of novel view synthesis on the four scenes of the tanks and temples dataset. The values in this table represent the mean loss over all test images. ADOP outperforms the other approach with and without test refinement (TR). In the latter, we estimate the exposure value and refine the camera pose of the test images.

| Method       | Train   | Playground | M60       | Lighthouse |
|--------------|---------|------------|-----------|------------|
|              | VGG ↓   | LPIPS ↓    | PSNR ↑    | VGG ↓      | LPIPS ↓    | PSNR ↑    | VGG ↓      | LPIPS ↓    | PSNR ↑    |
| Synsin + Unet| 706.0   | 0.3853     | 16.97     | 521.3      | 0.3198     | 21.81     | 564.3      | 0.2904     | 20.16     |
| Pulsar + Unet| 677.9   | 0.3418     | 17.78     | 661.2      | 0.3849     | 20.00     | 508.2      | 0.2403     | 21.42     |
| NPBG         | 817.5   | 0.4552     | 14.44     | 632.8      | 0.4110     | 19.47     | 741.2      | 0.4476     | 16.96     |
| NeRF++       | 857.5   | 0.5168     | 18.04     | 696.8      | 0.5292     | 22.24     | 700.5      | 0.4378     | 23.06     |
| SVS (half res.)| 633.1  | 0.323      | 17.43     | 516.6      | 0.3462     | 22.11     | 461.1      | 0.2440     | 23.74     |
| ADOP         | 422.6   | 0.1603     | 21.62     | 345.9      | 0.1594     | 25.00     | 342.8      | 0.1344     | 25.07     |
| ADOP w. TR   | 368.1   | 0.1438     | 23.19     | 318.7      | 0.1464     | 25.78     | 317.7      | 0.1189     | 25.84     |

ACM Trans. Graph., Vol. 41, No. 4, Article 99. Publication date: July 2022.
rendering, which natively supports the camera models, so they have been trained on the original distorted images.

The results (see Fig. 10) show that ADOP is able to produce convincing results on all tested scenes. Compared to the other methods, our renderings show less artifacts as well as preserve color, brightness, and sharpness better. For example, in the close-up view of the playground scene, the metal bars of the swing are most detailed in the ADOP rendering. In the M60 dataset, NPBG has similar visual results, but the quantitative evaluation (in Tab. 4) shows that our method still has the edge. In the last row of Tab. 4 we show the quantitative results of ADOP with test refinement (TR) enabled. TR compensates errors of the initial reconstruction by estimating the exposure value and camera pose of the test frames once training has been completed. This further improves the loss however note that even without TR our approach outperforms the state-of-the-art.

4.9 Fisheye Camera
Our system is able to learn with images originating from fisheye cameras without needing to undistort the projection, resulting in high quality neural fisheye renders (see Fig. 11). The office scene uses a LiDAR and SLAM system for poses and the point cloud, which can be noisy. As presented before, ADOP is able to correct this efficiently.

4.10 View Extrapolation
In Fig. 12 we show reconstructions from a view that is far from the input views, comparing the results of our system with SVS. In general, ADOP generates relatively good results in such cases compared to established methods, although this setup is generally difficult.

4.11 HDR Neural Rendering
When capturing outdoor scenes, it is almost unavoidable to use varying exposure to be able to adapt to the huge range of brightness values. To test the performance of our pipeline on such HDR scenes, we apply it to our Boat dataset. The camera was set to auto exposure, all other image related settings, such as aperture, ISO-Speed, and white balance were set constant for all frames. In Fig. 13, four samples from the dataset are shown, underneath the exposure value is plotted for every frame in the dataset. The difference between the smallest and largest EV is 8.7, which corresponds to a factor of $2^{8.7}$ = 426.67.

Novel view synthesis results of the Boat scene are shown in Fig. 14. For the images in the left column, uniform exposure was assumed, in the middle column the exposure value used by the tonemapper has been set to the real EV from the EXIF meta data and kept constant during training. Our system is then able to handle the high dynamic range and avoids splotchy artifacts.
We have presented ADOP, a novel fully differentiable neural point-based rendering pipeline that uses a set of calibrated images and an estimated point cloud of the scene as input. All stages are differentiable, so we can refine all input parameters to generate an improved output. By also including a photometric sensor model, we can handle input images with varying exposure and generate high dynamic range output.

Our experiments show that, due to the optimized input, we achieve superior rendering quality, especially for novel view synthesis. At the same time, the one-pixel point renderer is very fast. In combination with the fact that the improved consistency in the input allows us to use a simpler and thus faster neural network for reconstruction, we achieve real-time frame rates on a standard GPU, even for complex scenes.

https://github.com/darglein/ADOP

ACKNOWLEDGMENTS
Linus Franke was supported by the Bayerische Forschungsstiftung (Bavarian Research Foundation) AZ-1422-20.

REFERENCES
Agiisoft. 2021. Metashape. https://www.agisoft.com/
Marc Alexa, Markus Gross, Mark Pauly, Hanspeter Pfister, Marc Stamminger, and Matthias Zwicker. 2004. Point-based Computer Graphics. In ACM SIGGRAPH 2004 Course Notes, 7–es.
Kara-Alli Aliev, Artem Sevastopolsky, Maria Kolos, Dmitry Ulyanov, and Victor Lempitsky. 2020. Neural Point-Based Graphics. In Computer Vision—ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXII. 696–712. Springer.
Mario Botsch, Alexander Hormann, Matthias Zwicker, and Leif Kobbelt. 2005. High-quality surface splatting on today’s GPUs. In Proceedings Eurographics/EIEEE VGTC Symposium Point-Based Graphics, 2005. IEEE, 17–141.
Giang Bui, Truc Le, Brittany Morago, and Ye Duan. 2018. Point-Based Rendering Enhancement via Deep Learning. The Visual Computer 34, 6 (2018), 829–841.
Gaurav Chaurasia, Sylvain Duchene, Olga Sorkine-Hornung, and George Drettakis. 2013. Depth synthesis and local warps for plausible image-based navigation. ACM Transactions on Graphics (TOG) 32, 3 (2013), 1–12.
Wenzheng Chen, Huan Ling, Jun Gao, Edward Smith, Jaakko Lehtinen, Alec Jacobson, and Sanja Fidler. 2019. Learning to Predict 3D Objects With an Interpolation-Based Differentiable Renderer. Advances in Neural Information Processing Systems 32 (2019), 9609–9619.
Peng Dai, Yinda Zhang, Zhuwen Li, Shuaicheng Liu, and Bing Zeng. 2020. Neural Point Cloud Rendering via Multi-Plane Projection. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).
Paul Debevec, Yizhou Yu, and George D. Boshokov. 1998. Efficient view-dependent IBR with projective texture-mapping. In EG Rendering Workshop. Vol. 4.
Paul E Debevec and Jitendra Malik. 2008. Recovering High Dynamic Range Radiance Maps from Photographs. In ACM SIGGRAPH 2008 classes. 1–10.
Eric Enderton, Erik Sintorn, Peter Shirley, and David Luebke. 2010. Stochastic Transparency. IEEE transactions on visualization and computer graphics 17, 8 (2010), 2036–2047.
Jakoł Engel, Vladlen Koltun, and Daniel Cremers. 2017. Direct Sparse Odometry. IEEE transactions on pattern analysis and machine intelligence 40, 3 (2017), 611–625.
John Flynn, Ivan Neulander, James Phiblin, and Noah Snavely. 2016. DeepStereo: Learning to Predict New Views from the World’s Imagery. In Proceedings of the IEEE conference on computer vision and pattern recognition. 5515–5524.
Yaroslav Ganin, Daniil Konononok, Diana Sungatullina, and Victor Lempitsky. 2016. DeepWarp: Photorealistic Image Resynthesis for Gaze Manipulation. In European conference on computer vision. Springer, 311–326.
Kyle Genova, Forrester Cole, Aaron Maschinot, Daniel Arnaiz, and William T Freeman. 2018. Unsupervised Training for 3D Morphable Model Regression. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 8377–8386.
Daniel B Goldman. 2010. Vignette and Exposure Calibration and Compensation. IEEE transactions on pattern analysis and machine intelligence 32, 12 (2010), 2276–2288.
Michael D Grossberg and Shre K Nayyar. 2003. Determining the Camera Response from Images: What is Knowable? IEEE Transactions on pattern analysis and machine intelligence 25, 11 (2003), 1455–1467.
Jeffrey P Grossman and William J Dally. 1998. Point Sample Rendering. In Eurographics Workshop on Rendering Techniques. Springer, 181–192.

ACM Trans. Graph., Vol. 41, No. 4, Article 99. Publication date: July 2022.
Another application of our differentiable rendering pipeline is the alignment of camera images to point-clouds that were reconstructed by external devices. NavVis [NavVis 2021] provided us a dataset captured by the VLX mobile scanning platform. This platform consists of a high performance LiDAR scanner and four $5472 \times 3648$ pixel fisheye cameras. Their reconstruction software is able to combine multiple laser scans into a consistent point cloud and provides camera pose estimates for panorama generation and point cloud coloring. However, the image to point cloud registration is not perfect. Small errors during the SLAM-based tracking and vibrations due to the hand operation result in pose errors in the scale of millimeters. If we then use these poses for precise operation, such as reprojecting the color of neighboring source images into a target view, small ghosting artifacts can be observed (see Figure 16 center row). We train our neural rendering pipeline on this office dataset, which is composed of 688 images and 73M points, to synthesize all captured views. During training, our system also optimizes the camera pose of each frame. The refined poses are then used to reproject the colors into the same target frame as before (see Figure 16 bottom row). It can be seen that the ghosting artifacts are mostly eliminated and the synthesized view is much sharper. This experiment shows that the proposed method is able to perform pixel-perfect alignment of fisheye camera images to a point cloud of a LiDAR scanner. To our knowledge, no other available differentiable renderer can fulfill this task, as they assume a pinhole camera model or are not able to handle a point cloud with 73M points.

**B Runtime Performance**

Runtime performance has been a limiting factor for differentiable rendering systems in the past [Kato et al. 2020]. Most software rasterization techniques exceed the 100 ms barrier even for small scenes and render resolutions [Lassner and Zollhöfer 2021]. This limits their usefulness in real-world applications such as 3D reconstruction from high-resolution photographs. Currently, the two most performant differentiable rendering methods that are able to process point-cloud data are Synsin [Wiles et al. 2020] and Pulsar [Lassner and Zollhöfer 2021]. Synsin, which is the default point render engine of PyTorch3D [Ravi et al. 2020], splits each point to a disk and blends the $K$ nearest points of each pixel into the output image. Pulsar converts each point to a sphere and blends them with a similar approach as Soft Rasterizer [Liu et al. 2019a]. Both methods are fully differentiable, meaning that the point position and color can be optimized during rendering.

Table 5 shows the measured GPU frame-time for Synsin, Pulsar, our approach, and OpenGL’s default point rendering with GL_POINTS.
Table 5. Forward and backward render-time in milliseconds of a 1920 × 1080 image on a RTX 2080 Ti. In comparison to other differentiable renderers, our approach is around two magnitudes more efficient. The highlighted row is our approach with stochastic point discarding (see Section 3.1) that we use for scene refinement and novel-view synthesis.

| Geometry | # Layers | Forward | Backward |
|----------|----------|---------|----------|
| Synsin   | Disc Splats | 856.63  | 14.14    |
| Pulsar ($r = 0.005$) | Spheres | 31.20   | 3.62     |
| Pulsar ($r = 0.02$)  | Spheres | 36.81   | 3.02     |
| Pulsar ($r = 0.05$)  | Spheres | 53.29   | 3.09     |
| GL_POINTS | 1-Pixel Points | 0.3    | ×        |
| Ours     | 1-Pixel Points | 0.83   | 0.62     |
| Ours     | 1-Pixel Points | 1.4    | 0.77     |
| Ours + Stoc. Disc. | 1-Pixel Points | 1.28   | 0.66     |

# Points | 1,348,406 | 2,570,810 | 5,400,615 | 10,283,243

These timings only include the rasterization itself without the neural network and tonemapper described in the previous sections. For our method, we also include the rendering time of four layers in different resolutions. This is a more fair comparison to the other methods because all four layers are required for the neural rendering network. The output of Synsin and Pulsar is more complete and therefore a single layer can already be successfully used. The rightmost columns of Table 5 show the forward and backward render time of a 1920 × 1080 image for a point cloud with around 10M points. Both Synsin and Pulsar are not real-time capable at such dimensions with forward timings of 7342 ms and 209 ms respectively. Our approach takes two magnitudes less time than Pulsar with a combined rendering time of 3.65 ms for all four layers. This result is expected though because previous work has shown that software one-pixel point rendering can outperform hardware rasterization techniques [Schütz et al. 2021]. Point splatting and sphere rendering is inherently more complex because each point effects multiple output pixels.

If we enable stochastic discarding (see Section 3.1), the rendering performance is further increased. The largest gains are achieved for multi-resolution rendering of large point clouds. For example, generating a multi-layer image of the cloud with 10.4M points takes 19% less time with our stochastic discarding approach. However, if only a single image layer is required, the speedup due to stochastic discarding reduces to 3%, as points are only discarded if they fill less than one pixel in the output image. The pixels inside the low resolution layers are much larger and therefore more points are discarded. In comparison to native GL_POINTS rendering, our approach is only slightly slower (by about 26%). This is an impressive result, because we have implemented a three-pass blending approach with fuzzy depth-test as described in Section 3.1. The GL_POINTS reference implementation in Table 5 uses a single pass without blending and standard GL_LESS depth test.

Fig. 16. The initial camera pose estimates of the SLAM-System are slightly misaligned w.r.t. the LiDAR point cloud. Reprojecting the pixel color of several source views into a target view produces ghosting artifacts (center row). Our system is able to optimize the camera poses resulting in almost pixel perfect reprojections (bottom row).