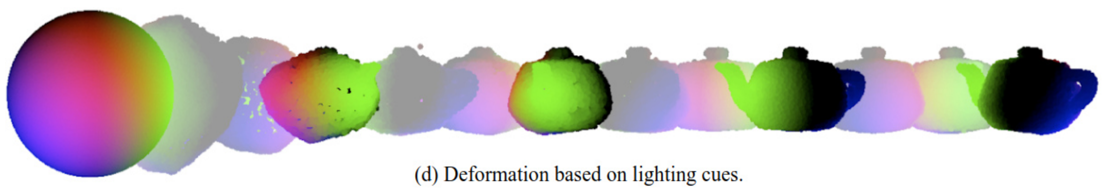
Pulsar: Efficient Sphere-based Neural Rendering

Christoph Lassner[1] Michael Zollhöfer[1]
[1]Facebook Reality Labs

(a) 3D reconstruction and neural shading.

(b) Sphere representation for (a).

(c) Viewpoint Synthesis.



(d) Deformation based on lighting cues.

Figure 1: Pulsar is an efficient sphere-based differentiable renderering module that is orders of magnitude faster than competing techniques, modular, and easy-to-use. It can be employed to solve a large variety of applications, since it is tightly integrated with PyTorch. Using a sphere-based representation, it is possible to not only optimize for color and opacity, but also for positions and radii (a, b, c). Due to the modular design, lighting cues can also be easily integrated (d).

Abstract

We propose Pulsar, an efficient sphere-based differentiable rendering module that is orders of magnitude faster than competing techniques, modular, and easy-to-use due to its tight integration with PyTorch. Differentiable rendering is the foundation for modern neural rendering approaches, since it enables end-to-end training of 3D scene representations from image observations. However, gradient-based optimization of neural mesh, voxel, or function representations suffers from multiple challenges, i.e., topological inconsistencies, high memory footprints, or slow rendering speeds. To alleviate these problems, Pulsar employs: 1) a sphere-based scene representation, 2) a modular, efficient differentiable projection operation, and 3) (optional) neural shading. Pulsar executes orders of magnitude faster than existing techniques and allows real-time rendering and optimization of representations with millions of spheres. Using spheres for the scene representation, unprecedented speed is obtained while avoiding topology problems. Pulsar is fully differentiable and thus enables a plethora of applications, ranging from 3D reconstruction to neural rendering.

1. Introduction

A differentiable rendering pipeline is the foundation for all modern neural rendering approaches that learn 3D scene representations based on image observations. Recently, neural rendering has empowered a large variety of applications, such as novel-view synthesis [22], facial reenactment [36], and 3D reconstruction [17]. Modern neural rendering can be broken up into three components: 1) a 3D neural scene representation, 2) a projection from 3D data to a consistent 2D representation (the *projection step*) and 3) processing the projected data using a statistical model, usually a neural network, to produce the final image (the *neural shading step*). This strategy combines the strengths of classical rendering and neural networks. Through the projection step, a consistent geometric representation of the scene is generated, while the neural shading step can produce realistic images through the use of the latest generative neural networks that can approximate complex natural image formation phenomena without having to explicitly model them.

Ideally, such a neural rendering approach can be jointly trained in an end-to-end fashion: a 3D representation of the scene is learned and sent through the projection and shading step. The resulting image can be compared to ground truth observations to inform an optimization process, not only to improve the generative model in the shading step, but also to jointly learn the representation of the scene and potentially unknown parameters of the projection step. This process requires the efficient computation of gradients through the complete pipeline in a scalable manner, such that high performance can be obtained even for the geometry of complex and detailed scenes rendered at real-world resolutions.

In this paper, we present *Pulsar*, an efficent, sphere-based, differentiable rendering module that is orders of magnitude faster than competing techniques, modular, and easy-to-use due to its tight integration with PyTorch. Pulsar aims to fulfill all mentioned requirements through a variety of measures, from the design of the scene representation down to low-level data-parallel optimizations, which lead to unprecedented speed for the forward and backward pass. First, we choose an entirely sphere-based representation of 3D data. Each sphere is parameterized by its position in space and its radius. In addition, each sphere has an assigned opacity and can have an arbitrary vector as payload, such as a color or a general latent feature vector. Image formation is based on a volumetric compositing schema that aggregates the payload in back-to-front order to form the final image or a screen space feature map. This makes it easy to handle point cloud data from 3D sensors directly, allows for the optimization of the scene representation without problems of changing topology (as they would exist for meshes) and is more efficient for rendering than recent approaches based on volumetric grids or fully-connected networks, since our representation, sparse by design, culls empty space. In addition, our sphere-based representation eliminates the need for acceleration structures, such as a k-d tree or octree, thus naturally can support dynamic scenes. Additionally, it leads to a well-defined, simple render and blending function that can be differentiated without approximation. We deliberately keep the illumination computations separate from the geometry projection step as it can be easily handled in a separate step. Lastly, we integrate Pulsar with the PyTorch [26] optimization framework to make use of auto-differentiation and ease the integration with deep learning models.

The strategy described above allows Pulsar to render and differentiate through the image formation for 3D scenes with millions of spheres on consumer graphics hardware. Up to one million spheres can be rendered and updated at real-time speed for an image resolution of 1024×1024 pixels (the time spent executing C++ code is less than $22\,\mathrm{ms}$ for rendering and less than $6\,\mathrm{ms}$ for gradient calculation). Pulsar supports a generalized pinhole and orthogonal camera model and computes gradients for camera parameters as well as to update the scene representation. We demonstrate that a large variety of applications can be successfully handled using Pulsar, such as 3D reconstruction, neural rendering, and viewpoint synthesis. Pulsar is open source and thus will enable researchers in the future to solve a large variety of research problems on their own. In summary, our main technical contributions are:

- A fast, general purpose, sphere-based, differentiable rendering module that is tightly integrated in PyTorch and enables end-to-end training of deep models with geometry and projection components.
- Pulsar executes orders of magnitude faster than existing techniques and allows real-time rendering and optimization of representations with millions of spheres.
- We demonstrate that a large variety of applications can be handled with Pulsar, such as 3D reconstruction, modeling realistic human heads, and novel view synthesis for scenes.

2. Related Work

We focus our discussion of related work on differentiable rendering and commonly used scene representations, such as textured meshes, voxel grids, (implicit) functions, and point-based representations. For a comprehensive review of neural rendering, we refer to the recent state of the art report on 'Neural Rendering' of Tewari et al. [35].

Differentiable Rendering Differentiable rendering can be understood as a subfield of *inverse graphics*, which has been a part of computer vision research since its early days [4]. For a summary of the features of current approaches, see Tab. 1. One of the seminal works on differentiable rendering of meshes, including lighting and textures, is OpenDR [20]. It is built on top of OpenGL and uses local Taylor expansions and filter operations to find gradients, excluding depth. OpenDR leverages existing OpenGL infrastructure, but introduces approximations and has a large overhead in running filtering and approximation operations. Neural Mesh Renderer (NMR) [10] renders meshes using a custom function to address object boundaries. Paparazzi et al. [14] is another mesh renderer that is implemented using image filtering. Pix2Vex [27] is a mesh renderer that uses soft blending of triangles. In the same spirit, Liu et al. [16] introduce a probabilistic map of mesh triangles. They use a soft z-buffer to obtain a differentiable representation. Their rendering function inspired our formulation. Tensorflow Graphics [40] is a differentiable rendering package for Tensorflow [1] with support for mesh geometry. Similarly, PyTorch3D [28] is a differentiable rendering package for PyTorch and initially focused on mesh rendering. A recent extension makes point-based rendering available and has been used for creating SynSin [42]. Pulsar executes orders of magnitude faster than these techniques.

method	objective	position update	depth update	normal update	occlusion	silhouette change	topology change
OpenDR	mesh	✓	✗	via position change	✗	✓	✗
NMR	mesh	✓	✗	via position change	✗	✓	✗
Paparazzi	mesh	limited	limited	via position change	✗	✗	✗
Soft Rasterizer	mesh	✓	✓	via position change	✓	✓	✗
Pix2Vex	mesh	✓	✓	via position change	✓	✓	✗
Tensorflow Graphics	mesh	✓	✓	via position change	✓	✓	✗
PyTorch3D	mesh / points	✓	✓	via position change	✓	✓	✓
DSS	points	✓	✓	✓	✓	✓	✓
Pulsar (ours)	spheres	✓	✓	via extra channels	✓	✓	✓

Table 1: Feature comparison of generic differentiable and modules (compare to [44], Tab. 1). DSS and PyTorch3D are the only other renderers that do not require a mesh-based geometry representation, facilitating topology changes. In contrast to DSS, Pulsar uses 3D spheres but without normals. Extra channels can be used to capture and optimize normal information.

Physically-based Differentiable Rendering Several renderers aim to be close to the underlying physical processes. Li et al. [3, 12] implement differentiable ray tracers to be able to compute gradients for physics-based rendering effects. These implementations explicitly model the image formation process in much greater detail, but are significantly slower in execution. Similarly, the Mitsuba 2 renderer [24] and Difftaichi [7] are physically-based differentiable renderers with slower execution times, but modelling the full image formation process including lighting and secondary rays. Whereas it would be possible to implement Pulsar using Difftaichi or Enoki (Mitsuba's autodiff framework), it would not be possible to implement many of Pulsars optimization strategies. In this work, we do not focus on physics-based approaches, since our aim is a fast differentiable module to empower neural rendering approaches that approximate natural image formation phenomena without having to explicitly model them.

Scene Representations There is a large variety of possible scene presentations from dense voxel grids [5, 19, 33, 39, 43], multi-plane images [21, 46], meshes [10, 16, 18, 20, 37], function-based representations [15, 22, 31, 34], to point-based representations (discussed in the next paragraph). In contrast to using explicit differentiable graphics engines, neural rendering can also be implemented solely through deep learning models. This is, for example, attempted in [11]. Implicit functions, such as signed distance fields (SDFs), are a popular representation for geometry. Recently, fully connected networks [25] have been used to learn SDFs. Liu et al. [17] optimize a signed distance function via differentiable sphere tracing. Similarly, Saito et al. [30] model humans through an implicit function. Zeng et al. [45] optimize a similar function using a differentiable renderer. Jiang et al. [9] is implementing differentiable rendering directly for SDFs, including lighting. RenderNet [23] is a CNN architecture with a projection unit. Tulsiani et al. [38] use a layered depth image representation and develop a differentiable renderer for optimizing this representation. Instead of prescribing a fixed input size or discretizing the underlying 3D scene, Sitzmann et al. [34] and Mildenhall et al. [22] represent the scene using the network structure and employ variants of ray-casting to reconstruct images from arbitrary viewpoints.

Point-based Representations Insafutdinov et al. [8] propose to work with differentiable point clouds. They train a CNN to predict the shape and pose of an object in 3D and use an orthographic projection and ray termination probabilities to obtain a differentiable representation. In contrast to our approach, their method is strongly limited in terms of resolution (they use 128×128 pixel image resolution and only up to 16k points in their paper); this is too low to solve real world tasks. Yifan et al. [44] propose a point-based representation with a position and normal parameterization. Each point is rendered as a small 'surface' with position, radius, and normal. In the examples shown in their paper, they use representations with up to 100k points and report orders of magnitude slower runtime than our approach (258 ms forward and 680 ms backward for an image of resolution 256×256 pixels). Lin et al. [13] define a renderer for point cloud generation, but only provide gradients for depth values. Roveri et al. [29] define a point-based differentiable projection module for neural networks that produces 'soft' depth images. Aliev et al. [2] propose to model room-scale point clouds with a deferred neural rendering step. SynSin [42] and the PyTorch3D point renderer follow a similar approach to ours, but are orders of magnitude slower. In addition, we employ a different blending function and enable the optimization of the sphere radius. Furthermore, they use only the first few points per pixel to determine the pixel colors. We have found this to leads to high frequency artifacts in complex scenes and thus allow for an unlimited number of spheres to contribute to the pixel color (or set a bound based on the minimum contribution) and use only the first few spheres for gradient propagation.

3. Method

We are interested in neural rendering approaches that learn a 3D scene representation from a set of N training im-

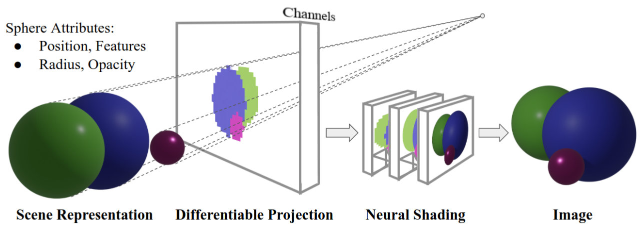


Figure 2: Visualization of the neural rendering pipeline. Pulsar enables a particularly fast differentiable projection step that scales to complex scene representations. The scene representation itself can be produced by a neural network. The channel information can be 'latent' and translated to RGB colors in a neural shading step.

ages $\mathcal{T} = \{(\mathbf{I}_i, \mathbf{R}_i, \mathbf{t}_i, \mathbf{K}_i)\}_{i=1}^N$. Here, the $\mathbf{I}_i \in \mathbb{R}^{H \times W \times 3}$ are image observations, the $\mathbf{R}_i \in \mathbb{R}^{3\times 3}$ are camera rotations, the $\mathbf{t}_i \in \mathbb{R}^3$ are camera translations, and the $\mathbf{K}_i \in \mathbb{R}^{3\times 3}$ are intrinsic camera parameters [6]. The neural rendering task can be split into three stages (see Fig. 2): 1) the scene representation (a traditional mesh model or potentially itself the result of a neural network), 2) a differentiable rendering operation, and 3) a neural shading module. Pulsar provides a particularly efficient differentiable rendering operation for scene representations that work with spheres as primitives.

3.1. Sphere-based Scene Representation

We represent the scene as a set $\mathcal{S} = \{(\mathbf{p}_i, \mathbf{f}_i, r_i, o_i)\}_{i=1}^M$ of M spheres with learned position $\mathbf{p}_i \in \mathbb{R}^3$, neural feature vector $\mathbf{f}_i \in \mathbb{R}^d$, radius $r_i \in \mathbb{R}$, and opacity $o_i \in \mathbb{R}$. All of these scene properties can be optimized through the differentiable rendering operation. The neural feature vector $\mathbf{f}_i \in \mathbb{R}^d$ encodes local scene properties. Depending on its use, it can represent surface color, radiance, or be an abstract feature representation for use in a neural network. If radiance is directly learned, our scene representation can be understood as an efficient and sparse way to store a neural radiance field [22] by only storing the non-empty parts of space. The explicit sphere-based scene representation enables us to make use of multi-view and perspective geometry by modeling the image formation process explicitly.

3.2. Efficient Differentiable Projection

Pulsar implements a mapping $\mathbf{F} = \mathcal{R}(\mathcal{S}, \mathbf{R}, \mathbf{t}, \mathbf{K})$ that maps from the 3D sphere-based scene representation $\mathcal{S}$ to a rendered feature image $\mathbf{F}$ based on the image formation model defined by the camera rotation $\mathbf{R}$, translation $\mathbf{t}$, and intrinsic parameters $\mathbf{K}$. $\mathcal{R}$ is differentiable with respect to $\mathbf{R}$, $\mathbf{t}$ and most parts of $\mathbf{K}$, i.e, focal length and sensor size.

Feature Aggregation The rendering operation $\mathcal{R}$ has to compute the channel values for each pixel of the feature image $\mathbf{F}$ in a differentiable manner. To this end, we propose a blending function that combines the channel information based on the position, radius, and opacity of the spheres that are intersected by the camera ray associated with each pixel. For a given ray, we associate a blending weight w_i with each sphere i:

$$w_i = \frac{o_i \cdot d_i \cdot \exp(o_i \cdot \frac{z_i}{\gamma})}{\exp(\frac{\epsilon}{\gamma}) + \sum_k o_k \cdot d_k \cdot \exp(o_k \cdot \frac{z_k}{\gamma})}. \tag{1}$$

Similar to Liu et al. [16], we choose a weighed softmax function of the sphere intersection depth z_i as the basis for our definition. We employ normalized device coordinates $z_i \in [0, 1]$ where 0 denotes maximum depth. A scaling factor γ is used to push the representation to be more rigorous with respect to depth. Small values, such as $\gamma = 1^{-5}$, lead to 'hard' blending, while large values, such as $\gamma = 1$, lead to 'soft' blending. Depending on the quantities that are optimized it makes sense to use different values for gamma. $\gamma = 1$ and $\gamma = 1^{-5}$ are the limits we allow to maintain numerical stability. The additional offset $\exp(\frac{\epsilon}{\gamma})$ is the 'weight' for the background color of the scene, for a fixed small constant ϵ. d_i is the normalized orthogonal distance of the ray to the sphere center. This distance, since always orthogonal to the ray direction, automatically provides gradients for the two directions that are orthogonal to the ray direction. We define $d_i = \min(1, \frac{||\vec{d_i}||_2}{R_i})$, where $\vec{d_i}$ is the vector pointing orthogonal from the ray to the sphere center. Like this, d_i becomes a linear scaling factor in $[0, 1]$.

It is non trivial to integrate opacity into Eq. 1 in a differentiable way, since it has to be 'soft'. Assuming there is a per sphere opacity value o_i, it could be integrated as a factor into the exponential function, or as another linear scaling factor. Similar to [45], we observe that integrating it only as a scaling factor within the exponential function, mainly interacting with the depth component, often leaves spheres visible in front of the background. On the other hand, using it only as a scaling factor outside the exponential function, mainly interacting with the distance component, makes depth 'override' opacity in most cases and does not lead to appropriate results. Using it in both places is a feasible and numerically stable solution.

Data-parallel Implementation Pulsar is implemented in C++ and CUDA C as a PyTorch extension to make use of the processing power of modern GPUs and naturally integrate the rendering step with machine learning models. We found the following points important to achieve high performance: **A representation with a fast, closed form intersection calculation. The commutativity of the blending function**, which enables pixel-wise thread-collaboration when finding intersections. Additionally, it allows for early stopping. **Splitting the full task carefully**

into sub-tasks and choosing the right data-parallel implementation for each sub-task allows us to use hardware threads efficiently and with advantageous memory access patterns. For example, parallelizing over the spheres can be beneficial because of sharing of information for the evaluation of Eq. 1. However, this approach requires synchronization between threads when writing to the same pixel, which obliterates performance. The alternative is parallelizing over the pixels. In this case, it is important to exploit spatial closeness between pixels during the search for relevant spheres. To this end, we propose a tile-based acceleration structure and cooperative filtering across hardware levels for filtering. An additional benefit is that the rendering speed is largely independent of sphere sizes. **Memory layout** has a very high impact on performance. For example, we managed to encode all information required for a sphere intersection into 8 bytes. One cache line on NVIDIA Turing GPUs holds $256 = 8 \times 32$ bytes and has 32 threads in a warp—meaning that 32 sphere intersections can be performed in a collaborating warp in parallel. For a detailed discussion, we refer to the supplemental material.

3.3. Neural Shading

The task of the neural shading network $\mathcal{N}$ is to convert the 2D screen space feature map $\mathbf{F}$ to the output color image using a pixel generator network that learns the feature-to-color mapping. In the tests where we apply neural shading, we use either a convolutional U-Net or a per-pixel one-by-one convolutional network. The higher the capacity of the shading network, the more the approach can overfit. It is important to find the right trade-off based on the desired application and the available training data.

3.4. End-to-end Optimization

We find the best sphere-based neural 3D scene representation $\mathcal{S}^*$ and neural shading model $\mathcal{N}(\bullet; \Theta_s^*)$ through gradient-based optimization using Pulsar. We solve the following end-to-end optimization problem:

$$\mathcal{S}^*, \Theta_s^* = \arg\min_{\mathcal{S}, \Theta_s} \sum_{i=0}^{N} \left|\left| \mathbf{I}_i - \mathcal{N}(\mathcal{R}(\mathcal{S}, \mathbf{R}_i, \mathbf{t}_i, \mathbf{K}_i); \Theta_s) \right|\right|_1 .$$

Note that this optimization finds all parameters based on the 2D image observations without 3D supervision. Pulsar provides the efficient differentiable projection $\mathcal{R}$, while all other stages are implemented in a modular manner using PyTorch. We use ADAM in all experiments to solve this optimization problem. Whereas this is one straightforward application of the proposed rendering component, it can be used in a variety of settings and integrated in deep learning pipelines in a straightforward and modular way.

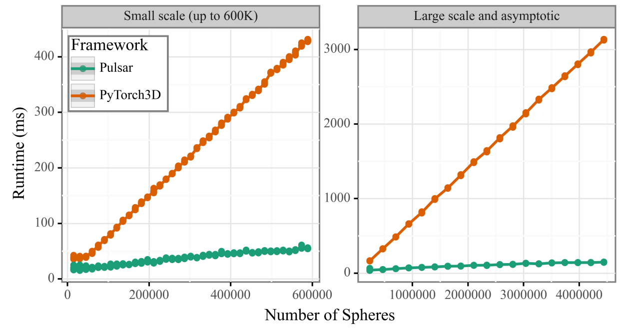


Figure 3: Scaling behavior for PyTorch3D and Pulsar for different numbers of spheres. Whereas PyTorch3D scales almost linearly in terms of number of spheres, Pulsar employs early-stopping and other optimization techniques to reach much better scaling behavior. Benchmarks performed on an NVIDIA RTX 2070 GPU at 1024×1024 resolution.

4. Results

In the following, we compare to other differentiable rendering methods in terms of training and test time performance and demonstrate the power and simplicity of the modular implementation of Pulsar as a PyTorch module.

Runtime Performance Rendering speed is even more important for differentiable and neural rendering than for traditional rendering, because it effectively limits the resolution of the images and scene representations that can be processed: the scene is not processed only 'once' in a forward pass, but continuously within an optimization loop. Optimizations with millions of spheres, as presented later in this paper, are prohibitively slow to perform with other renderers. To illustrate this, we compare Pulsar to a large variety of state-of-the-art neural/diff. rendering modules in terms of runtime on two scenes of varying complexity.

Pulsar outperforms all current state-of-the-art approaches, including PyTorch3D, by a large margin (see Tab. 2), in some comparisons more than two orders of magnitude and at least factor five. We continued performing measurements for the closest contenders, *PyTorch3D (points)* and Pulsar, to analyze the asymptotic behavior. The other frameworks generally were too slow to create meaningful comparisons on this scale. You can find the results in Fig. 3. For PyTorch3D, we used `points_per_pixel=5` to achieve a close match of conditions. The PyTorch3D sphere renderer already reaches a runtime of 400 ms below 500K spheres, whereas Pulsar remains still below this value at the maximum benchmarked amount of spheres at around 4.4M spheres. Apart from good scaling behavior in terms of number of spheres, we also observe good scaling behavior in terms of image size. For 4K image resolution with 4.4M spheres, we still measure execution times of less than 400 ms depending on GPU and scene: we measure 387 ms for 4.4M spheres on an NVIDIA RTX 2070 GPU) with moderate memory requirements (3500 MB).

method	number of points	number of faces	avg. forward time in ms	avg. backward time in ms
Soft Rasterizer	15 099	29 924	285	294
DSS	15 099	n.a.	215	179
PyTorch3D (mesh)	15 099	29 924	104	80
PyTorch3D (points) / SynSin	15 099	n.a.	34	2
pulsar	15 099	n.a.	14	1
pulsar (CUDA only)	15 099	n.a.	3	1
Soft Rasterizer	233 872	467 848	5032	5356
DSS	233 872	n.a.	3266	3690
PyTorch3D (mesh)	233 872	467 848	222	105
PyTorch3D (points) / SynSin	233 872	n.a.	112	3
Pulsar (Ours)	233 872	n.a.	**21**	**2**
Pulsar (CUDA only)	233 872	n.a.	**9**	**1**

Table 2: Runtime performance comparison of state-of-the-art differentiable rendering methods with PyTorch integration. For Pulsar, we measure the performance using the full Python interface (as for the other renderers) as well as the runtime of the CUDA kernel. *PyTorch3D (points)* uses a fixed point size for all points and the runtime does not scale well for larger point sizes. Pulsar's runtime is largely sphere size agnostic and scales favorably with resolution and number of spheres. For example, for 1 million spheres we still measure execution times of less than 33ms (19ms in CUDA) forward and 11ms (4.7ms in CUDA) backward. All times are measured on an NVIDIA RTX 2080 GPU at 1000×1000 image resolution.

```
1  import torch; torch.manual_seed(1)
2  from pulsar import Renderer
3  n_spheres = 10
4  # We create a renderer for a 1024x1024 image.
5  renderer = Renderer(1024, 1024, n_spheres)
6  pos = torch.rand(n_spheres, 3) * 10.0
7  pos[:, 2] += 25.0; pos[:, :2] -= 5.0
8  col = torch.rand(n_spheres, 3)
9  rad = torch.rand(n_spheres)
10 cam = torch.tensor(
11   #------t------  ------R------   f    s
12   [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 5.0, 2.0],
13   dtype=torch.float32)
14 image = renderer(pos, col, rad, cam,
15   gamma=0.1, max_depth=45.0)
16 # L1 loss, assuming `target` is an image.
17 # loss = (image - target).abs().sum()
18 # loss.backward()
19 # Use any PyTorch optimizer for optimization.
```

Listing 1: Full code example to render a minimal scene with 10 random spheres. The loss computation is shown in ll. 16 and 17 and can be used in any PyTorch optimization loop. f and s denote focal length and sensor width.

Ease of Use Lst. 1 shows a full example for generating and rendering a scene representation in only nine lines of code. The resulting image contains ten spheres, randomly placed and colored. On construction, the renderer creates buffer structures, for which the image size and the maximum amount of spheres to render must be known (l. 5). The scene representation is encoded in an intuitive way that can be integrated into a `nn.Module` structure or generated through other operations (ll. 6-8). The camera parameters are encoded in an optimization-friendly format (Pulsar accepts 6D rotation vectors [47], too) so that the camera parameter vector can be used for gradient updates (l. 9). In l. 13, the forward function generates an image tensor from the scene description. This tensor can be used for further computations, for example neural shading, or directly to define a loss (illustrated in ll. 16 and 17). The operations are automatically registered with PyTorch's autograd system.

5. Applications

We demonstrate several applications in 3D reconstruction and neural rendering to show the versatility of Pulsar in various optimization settings. In particular, we show how Pulsar makes it possible to attack different classical computer vision tasks in a straightforward way with its ability to use a highly detailed representations and high quality gradients. In all experiments we optimize or *reconstruct* appearance and geometry solely using a photometric ℓ_1-loss. We order the experiments by complexity, and in the last ones we use several millions of spheres during the optimization. These experiments would be prohibitively slow to perform with other differentiable rendering methods.

5.1. 3D Reconstruction

Silhouette-based 3D Reconstruction Pulsar enables 3D reconstruction of objects based on silhouettes. We demonstrate results on an example scene of SoftRas [16] that provides 120 views of an airplane, see Fig. 4(a). To work with our sphere-based representation, we place spheres at all vertices of the mesh SoftRas employs for initialization. SoftRas uses an image resolution of size 64×64, which pushes the size of each sphere to the lower limits in terms of pixel size. Instead of the more intricate and computationally complex IOU, Laplacian, and flattening losses that are required in SoftRas, we solely employ a photometric ℓ_1-loss with respect to the ground truth silhouettes. SoftRas requires these additional losses to keep the mesh surface consistent. In contrast, we can move spheres without taking surface topology into account. For a low number of spheres and small resolution SoftRas is faster, but Pulsar scales much better to real world scenarios (see Tab. 2).

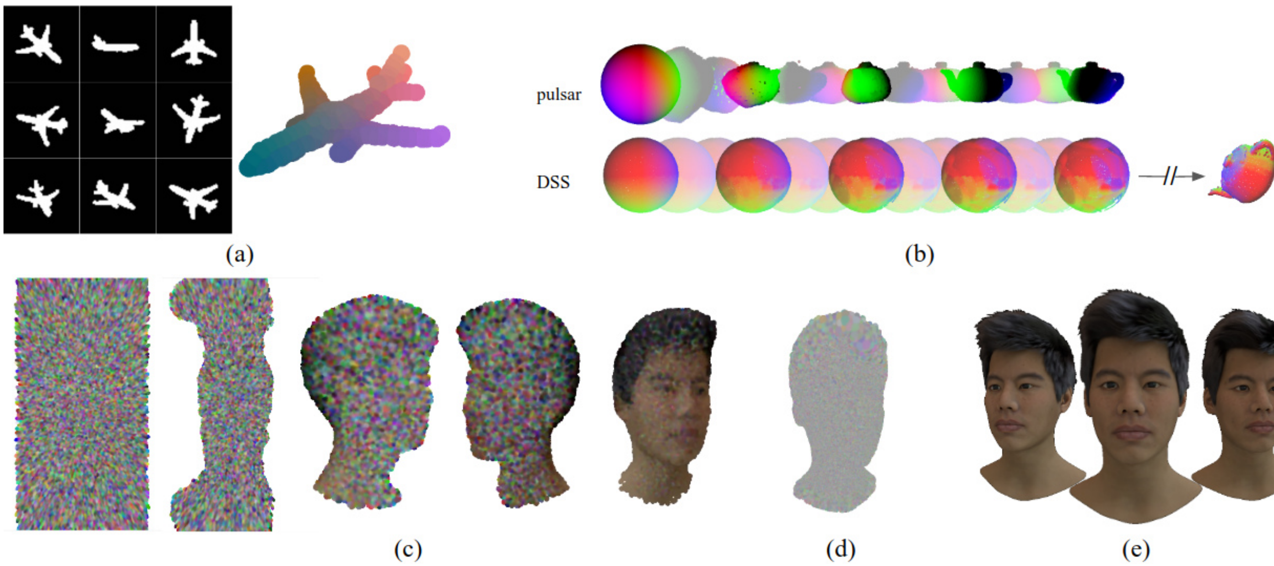


Figure 4: 3D reconstruction with Pulsar with up to 400k spheres. **(a)** Silhouette-based deformation reconstruction (c.t. [16]); 1352 spheres, 64×64. **(b)** Reconstruction with lighting cues and comparison with DSS [44]; 8003 spheres, 256×256. Pulsar finishes the reconstruction after 31 s, whereas DSS finishes after 1168 s. **(c)** Reconstruction steps of a 3D head model in 73 s; 400k spheres, 800×1280. 80 images with random azimuth and elevation are used. **(d)** Initialized features for training the neural shading model for (e). **(e)** Neural rendering results of a pix2pixHD [41] model based on this geometry.

Illumination Optimization We reproduce an experiment from DSS [44] that includes an illumination model, see Fig. 4(b). To this end, we implement diffuse shading with parallel light sources as a separate stage. This highlights the versatility of a dedicated geometry projection step and demonstrates how easy it can be combined with additional refinement models. Similar to DSS, we use 144 cameras, selected at random azimuth and elevation angles, with a constant distance to the object center. In this experiment, 300 optimization steps suffice for Pulsar to reach convergence. Using Pulsar, we complete the optimization in 31 seconds, whereas DSS requires more complex losses and almost 20 minutes to converge after 477 steps; Pulsar is more than one order of magnitude faster.

Detailed 3D Reconstruction Pulsar can go far beyond the number of spheres and image resolutions in the previous examples. We demonstrate this in the following experiments: first, by demonstrating a multi-stage pipeline for reconstructing a head model, then by working on several examples of the NeRF dataset [22]. In this experiment, we reconstruct a head model with realistic hair and eyes at high resolution, see Fig. 4(c), from 100 images of resolution 800×1280 pixels. We initialize a volume with 400k randomly distributed spheres and optimize for a coarse head model in only 73 s. We eliminate spheres if their color converges towards the background color or if they are not visible (spheres that do not receive gradient updates from any viewpoint). This results in a hull representation of the head with a thickness of several centimeters. After the optimization and cleanup, a model consisting of approx. 20k spheres remains. Next, we increase the number of spheres three times through subdivision: We refine each sphere with 12 spheres with radius $\sqrt{2} \cdot r$, where r is the previous radius, and place them in a face-centered cubic packing scheme, re-optimize and prune. The final model has 130k spheres, see Fig. 4(d). The refinement finishes in 37 minutes and temporarily produces models with up to 1.6 million spheres.

Neural Shading To showcase the potential of the proposed pipeline, we combine the learned head model, see Fig. 4(d), with a neural shading network for modelling the face in high resolution. As architecture, we used an off-the-shelf Pix2PixHD [41] design and used 15 feature channels. For the neural network training, we employ a photometric ℓ_1-loss, a perceptual loss [32], and an adversarial loss, as is standard for Pix2PixHD. We experimented with different number of training images: with 80 images in the training set we can already obtain a reconstruction that interpolates well between perspectives, but still with a visible loss in detail. With more than 320 images there's hardly any perceptual difference between training and validation results visible (see Fig. 4(e)). The model produces compelling results that can be rendered in near real time on consumer hardware (we achieve 30+ FPS for the geometry projection step and neural shading takes 37 ms).

5.2. Neural Rendering

Novel-view Synthesis on Real Data We perform novel-view synthesis experiments on examples of the NeRF

(a) (b) (c) (d)

Figure 5: High-resolution scene representation view synthesis and reconstruction examples with 1M and more spheres; scenes from the NeRF dataset [22]. **(a)** Test view of the 'flower' scene; 2.1M/810K spheres; 1008×756. **(b)** Test view of the 'fern' scene; 2.6M spheres; 1008×756. **(c)** Test view of the 'chair' scene with two different virtual viewpoints and shared per-pixel fully-connected shading model; 5.5M/509K spheres, 1600×1600. Note the viewpoint-dependent shading effects on the chair cover. **(d)** 360 degree views of the 'chair' model. X/Y spheres are before/after optimization.

dataset [22]. In the first example, see Fig. 5(a), we show that pure sphere geometry is sufficient to represent complex, real-world scenes. We initialize the scene by filling the volume in front of the camera uniformly with 2.1M spheres, with increasing radius according to depth. We use Pulsar to jointly optimize all sphere properties (*i.e.*, position, radius, color and opacity) based on an ℓ_1-loss using the Adam optimizer. We apply a threshold to spheres with low opacity and use an opacity-depth regularizer with the energy $-z_i \cdot O_i$ to encourage spheres to move to the right scene depth. The optimization runs in 20 minutes on an NVIDIA V100 GPU. The second example, see Fig. 5(b), shows a similar reconstruction on the 'fern' scene. We start the optimization with 5.5M random spheres and optimize across three scales to account for the details in the fern leafs.

View-dependent Shading We show separation into geometry projection and neural shading under challenging conditions, see Fig. 5(c-d). For this, we consider the synthetic 'chair' scene of the NeRF dataset, which has 200 views. The surface is highly textured and changes its appearance dramatically (satin cover) depending on the viewing angle. We start the optimization from 5.5M randomly initialized spheres and work in double image resolution to capture all texture details. We add a simple fully-connected model that is shared across all pixels to optimize for view-dependent appearance and condition on a per-pixel view direction. Gradients are back-propagated through the shading model to optimize the channel information. Even such a small model captures the viewpoint dependent effects well.

6. Limitations

While we have demonstrated high performance rendering and optimization of complex scenes with millions of spheres, our approach is still subject to a few limitations that can be addressed in future work: 1) It is challenging to compute gradients with respect to the position and radius of spheres that are smaller than a few pixel in screen space. We address this by explicitly handling this case in the rendering pipeline and by limiting the gradient computation for these spheres to the feature channels and opacity. In this way, they can be pruned through the opacity optimization and we prevent noisy gradients from leaking into the position- or radius-models. By finding better ways to handle these cases, we could obtain even better results. 2) While our module is highly flexible and can be easily integrated with arbitrary PyTorch training loops, it is currently not programmable. The CUDA kernels are highly optimized and contain the blending function and its symbolic derivative. This means that changing the function requires explicitly modifying the CUDA kernels, which is time consuming and error prone. A programmable shader language in combination with auto-differentiation could alleviate this problem while maintaining the high performance of hand written CUDA code.

7. Conclusion

We presented Pulsar, an efficient sphere-based differentiable rendering module. Its architecture builds on recent insights in the fields of differentiable rendering and neural networks and makes deliberate choices to limit complexity in the projection step to obtain high speed and scalability. Pulsar executes orders of magnitude faster than existing techniques and for the first time enables real-time rendering and optimization of representations with millions of spheres. We demonstrated its performance and flexibility on a large variety of applications ranging from 3D reconstruction to general neural rendering. Pulsar is open-source software, modular, and easy-to-use due to its tight integration with PyTorch. Through its performance and accessibility, we hope that Pulsar will enable researchers to explore new ideas that were out of reach before.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Kara-Ali Aliev, Dmitry Ulyanov, and Victor Lempitsky. Neural point-based graphics. *arXiv preprint arXiv:1906.08240*, 2019.

[3] Dejan Azinovic, Tzu-Mao Li, Anton Kaplanyan, and Matthias Nießner. Inverse path tracing for joint material and lighting estimation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2447–2456, 2019.

[4] Bruce Guenther Baumgart. Geometric modeling for computer vision. Technical Report AI Lab Memo AIM-249, Stanford University, 10 1974.

[5] Christopher B Choy, Danfei Xu, JunYoung Gwak, Kevin Chen, and Silvio Savarese. 3d-r2n2: A unified approach for single and multi-view 3d object reconstruction. In *European conference on computer vision*, pages 628–644. Springer, 2016.

[6] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, New York, NY, USA, 2 edition, 2003.

[7] Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. Difftaichi: Differentiable programming for physical simulation. *arXiv preprint arXiv:1910.00935*, 2019.

[8] Eldar Insafutdinov and Alexey Dosovitskiy. Unsupervised learning of shape and pose with differentiable point clouds. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31 (NIPS 2018)*, pages 2804–2814, Montréal, Canada, 2018. Curran Associates.

[9] Yue Jiang, Dantong Ji, Zhizhong Han, and Matthias Zwicker. Sdfdiff: Differentiable rendering of signed distance fields for 3d shape optimization. *arXiv preprint arXiv:1912.07109*, 2019.

[10] Hiroharu Kato, Yoshitaka Ushiku, and Tatsuya Harada. Neural 3d mesh renderer. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3907–3916, 2018.

[11] Tejas D Kulkarni, William F Whitney, Pushmeet Kohli, and Josh Tenenbaum. Deep convolutional inverse graphics network. In *Advances in neural information processing systems*, pages 2539–2547, 2015.

[12] Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. Differentiable monte carlo ray tracing through edge sampling. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)*, 37(6):222:1–222:11, 2018.

[13] Chen-Hsuan Lin, Chen Kong, and Simon Lucey. Learning efficient point cloud generation for dense 3d object reconstruction. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[14] Guilin Liu, Duygu Ceylan, Ersin Yumer, Jimei Yang, and Jyh-Ming Lien. Material editing using a physically based rendering network. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2261–2269, 2017.

[15] Lingjie Liu, Jiatao Gu, Kyaw Zaw Lin, Tat-Seng Chua, and Christian Theobalt. Neural sparse voxel fields, 2020.

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

[17] Shaohui Liu, Yinda Zhang, Songyou Peng, Boxin Shi, Marc Pollefeys, and Zhaopeng Cui. Dist: Rendering deep implicit signed distance function with differentiable sphere tracing. *arXiv preprint arXiv:1911.13225*, 2019.

[18] Stephen Lombardi, Jason Saragih, Tomas Simon, and Yaser Sheikh. Deep appearance models for face rendering. *ACM Trans. Graph.*, 37(4), July 2018.

[19] Stephen Lombardi, Tomas Simon, Jason Saragih, Gabriel Schwartz, Andreas Lehrmann, and Yaser Sheikh. Neural volumes: Learning dynamic renderable volumes from images. *ACM Trans. Graph.*, 38(4), July 2019.

[20] Matthew M. Loper and Michael J. Black. OpenDR: An approximate differentiable renderer. In *Computer Vision – ECCV 2014*, volume 8695 of *Lecture Notes in Computer Science*, pages 154–169. Springer International Publishing, Sept. 2014.

[21] Ben Mildenhall, Pratul P. Srinivasan, Rodrigo Ortiz-Cayon, Nima Khademi Kalantari, Ravi Ramamoorthi, Ren Ng, and Abhishek Kar. Local light field fusion: Practical view synthesis with prescriptive sampling guidelines. *ACM Transactions on Graphics (TOG)*, 2019.

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

[23] Thu H Nguyen-Phuoc, Chuan Li, Stephen Balaban, and Yongliang Yang. Rendernet: A deep convolutional network for differentiable rendering from 3d shapes. In *Advances in Neural Information Processing Systems*, pages 7891–7901, 2018.

[24] Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. Mitsuba 2: A retargetable forward and inverse renderer. *Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 38(6), Dec. 2019.

[25] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 165–174, 2019.

[26] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming

Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[27] Felix Petersen, Amit H Bermano, Oliver Deussen, and Daniel Cohen-Or. Pix2vex: Image-to-geometry reconstruction using a smooth differentiable renderer. *arXiv preprint arXiv:1903.11149*, 2019.

[28] Nikhila Ravi, Jeremy Reizenstein, David Novotny, Taylor Gordon, Wan-Yen Lo, Justin Johnson, and Georgia Gkioxari. Pytorch3d. https://github.com/facebookresearch/pytorch3d, 2020.

[29] Riccardo Roveri, Lukas Rahmann, Cengiz Oztireli, and Markus Gross. A network architecture for point cloud classification via automatic depth images generation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4176–4184, 2018.

[30] Shunsuke Saito, Zeng Huang, Ryota Natsume, Shigeo Morishima, Angjoo Kanazawa, and Hao Li. Pifu: Pixel-aligned implicit function for high-resolution clothed human digitization. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2304–2314, 2019.

[31] Katja Schwarz, Yiyi Liao, Michael Niemeyer, and Andreas Geiger. Graf: Generative radiance fields for 3d-aware image synthesis. *arXiv preprint arXiv:2007.02442*, 2020.

[32] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[33] V. Sitzmann, J. Thies, F. Heide, M. Nießner, G. Wetzstein, and M. Zollhöfer. Deepvoxels: Learning persistent 3d feature embeddings. In *Proceedings of Computer Vision and Pattern Recognition (CVPR 2019)*, 2019.

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

[35] A. Tewari, O. Fried, J. Thies, V. Sitzmann, S. Lombardi, K. Sunkavalli, R. Martin-Brualla, T. Simon, J. Saragih, M. Nießner, R. Pandey, S. Fanello, G. Wetzstein, J.-Y. Zhu, C. Theobalt, M. Agrawala, E. Shechtman, D. B. Goldman, and M. Zollhöfer. State of the art on neural rendering. *EUROGRAPHICS 2020*, 39(2), 2020.

[36] Justus Thies, Michael Zollhöfer, and Matthias Nießner. Deferred neural rendering: Image synthesis using neural textures. *ACM Trans. Graph.*, 38(4), July 2019.

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

[38] Shubham Tulsiani, Richard Tucker, and Noah Snavely. Layer-structured 3d scene inference via view synthesis. In *Proc. ECCV*, 2018.

[39] Shubham Tulsiani, Tinghui Zhou, Alexei A Efros, and Jitendra Malik. Multi-view supervision for single-view reconstruction via differentiable ray consistency. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2626–2634, 2017.

[40] Julien Valentin, Cem Keskin, Pavel Pidlypenskyi, Ameesh Makadia, Avneesh Sud, and Sofien Bouaziz. Tensorflow graphics: Computer graphics meets deep learning. 2019.

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

[42] Olivia Wiles, Georgia Gkioxari, Richard Szeliski, and Justin Johnson. Synsin: End-to-end view synthesis from a single image. *arXiv preprint arXiv:1912.08804*, 2019.

[43] Jiajun Wu, Chengkai Zhang, Tianfan Xue, Bill Freeman, and Josh Tenenbaum. Learning a probabilistic latent space of object shapes via 3d generative-adversarial modeling. In *Advances in neural information processing systems*, pages 82–90, 2016.

[44] Wang Yifan, Felice Serena, Shihao Wu, Cengiz Öztireli, and Olga Sorkine-Hornung. Differentiable surface splatting for point-based geometry processing. *ACM Transactions on Graphics (TOG)*, 38(6):1–14, 2019.

[45] Huang Zeng, Yuanlu Xu, Christoph Lassner, Hao Li, and Tony Tung. Arch: Animatable reconstruction of clothed humans. *arXiv preprint arXiv:2004.04572*, 2020.

[46] Tinghui Zhou, Richard Tucker, John Flynn, Graham Fyffe, and Noah Snavely. Stereo magnification: Learning view synthesis using multiplane images. *arXiv preprint arXiv:1805.09817*, 2018.

[47] Yi Zhou, Connelly Barnes, Jingwan Lu, Jimei Yang, and Hao Li. On the continuity of rotation representations in neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5745–5753, 2019.