Modular Primitives for High-Performance Differentiable Rendering

SAMULI LAINE, NVIDIA
JANNE HELLSTEN, NVIDIA
TERO KARRAS, NVIDIA
YEONGHO SEOL, NVIDIA
JAAKKO LEHTINEN, NVIDIA and Aalto University
TIMO AILA, NVIDIA

We present a modular differentiable renderer design that yields performance superior to previous methods by leveraging existing, highly optimized hardware graphics pipelines. Our design supports all crucial operations in a modern graphics pipeline: rasterizing large numbers of triangles, attribute interpolation, filtered texture lookups, as well as user-programmable shading and geometry processing, all in high resolutions. Our modular primitives allow custom, high-performance graphics pipelines to be built directly within automatic differentiation frameworks such as PyTorch or TensorFlow. As a motivating application, we formulate facial performance capture as an inverse rendering problem and show that it can be solved efficiently using our tools. Our results indicate that this simple and straightforward approach achieves excellent geometric correspondence between rendered results and reference imagery.

CCS Concepts: • **Computing methodologies** → **Rasterization**; **Shape inference**.

Additional Key Words and Phrases: Differentiable rendering, rasterization, motion capture.

ACM Reference Format:
Samuli Laine, Janne Hellsten, Tero Karras, Yeongho Seol, Jaakko Lehtinen, and Timo Aila. 2020. Modular Primitives for High-Performance Differentiable Rendering. *ACM Trans. Graph.* 39, 6, Article 194 (December 2020), 14 pages. https://doi.org/10.1145/3414685.3417861

1 INTRODUCTION

Differentiable rendering is a fundamental building block in machine learning of 3D geometry. Typically training data is available only as images, and finding a corresponding 3D representation requires *analysis by synthesis*, i.e., rendering candidate images, computing the loss based on training and candidate images, and propagating the errors back to 3D positions and other scene attributes. Many classical computer graphics and vision problems including the estimation of reflectance, geometry, lighting, and camera parameters can be cast into this *inverse rendering* framework [Patow and Pueyo 2003].

Much of modern machine learning makes use of first order (gradient-based) optimization techniques implemented using backpropagation. From a computational point of view, the explosive growth in model sizes and capabilities has, for a large part, relied on the availability of primitive operations that allow massively parallel and coherent execution of both the forward and backward (gradient) passes, allowing highly complex computation graphs to be built out of them. However, the majority of effort in creating efficient primitive operations has focused on operating on densely-sampled data stored in multidimensional regular grids. These kind of operations alone are not sufficient for 3D rendering because the mapping from a scene representation to pixel values is highly irregular and dynamic. As such, specialized differentiable rendering algorithms have emerged for solving two problems:

(1) *Forward pass:* Given the factors that affect the shape and appearance of a 3D scene, render a 2D image; and
(2) *Backward pass:* Given the gradient of a loss function defined on the output image pixels, compute the gradient of the loss with respect to the input shape and appearance factors.

This interface is universally used by autodifferentiation frameworks such as PyTorch and TensorFlow, and it allows 3D rendering to be used as a building block in a complex model that is trained by modern stochastic first order optimization techniques.

Despite its long history, differentiable rendering can be considered a nascent field due to the recent proliferation of algorithms and applications. Most previous research is targeted towards a specific use case (e.g., pose or shape estimation), and is typically only evaluated on downstream tasks as part of a larger machine learning system [Chen et al. 2019; Kato et al. 2018; Liu et al. 2019]. These specific use cases and data sets allow optimizations and design choices that do not scale to other uses. For example, low geometric complexity may make it acceptable to not parallelize over triangles, but this quickly backfires on a larger scene; single objects viewed in a vacuum may enable one to disregard the effects of mutual occlusion between triangles when computing gradients, but this approximation is untenable in a scene with non-trivial depth complexity.

Another parallel line of research studies differentiable physically-based light transport simulation that models complex effects such as area light sources and indirect illumination [Li et al. 2018; Loubet et al. 2019; Nimier-David et al. 2019]. Built on Monte Carlo sampling, these methods seek the best attainable image quality and accuracy at the cost of longer rendering times.

We build on the rich literature on real-time graphics systems that has long sought efficient solutions for managing the complex, dynamic mapping between world points and image pixels, and has delivered extremely efficient and practical hardware implementations. In particular, we seek to formulate and implement a differentiable rendering system that makes use of these pipelines to maximal

Authors' addresses: Samuli Laine, NVIDIA, slaine@nvidia.com; Janne Hellsten, NVIDIA, jhellsten@nvidia.com; Tero Karras, NVIDIA, tkarras@nvidia.com; Yeongho Seol, NVIDIA, yseol@nvidia.com; Jaakko Lehtinen, NVIDIA, Aalto University, jlehtinen@nvidia.com; Timo Aila, NVIDIA, taila@nvidia.com.

Table 1. Comparison of characteristics of selected differentiable rendering systems.

	OpenDR [2014]	NMR [2018]	SoftRas [2019]	DIB-R [2019]	Li et al. [2018]	Mitsuba2 [2019]	**Our**
Performance*	✔	✗	✗	✗	✖	✖	✔
Scalability	✔	✖	✖	✖	✔	✔	✔
Flexibility	✖	✖	✖	✖	✔	✔	✔
Antialiasing	✖	✖	✗	✖	✔	✔	✔
Occlusion	✔	✔	✖	✗	✔	✔	✔
Gradients	✖	✖	✔	✗	✔	✗	✔
Noise-free	✔	✔	✔	✔	✖	✖	✔
No tuning	✔	✔	✖	✖	✔	✗	✔
GI	✖	✖	✖	✖	✔	✔	✖

**Performance* considers suitability to intensive optimization such as in deep learning; *Scalability* refers to performance with respect to surface tessellation and image resolution; *Flexibility* is whether the system is designed to support arbitrary shading; *Antialiasing* requires that geometric edges are smoothed in the result image; *Occlusion* considers if geometrically obscured surfaces are guaranteed to not affect the resulting image; *Gradients* refers to the correctness of gradients with respect to rendered image; *Noise-free* systems do not rely on random sampling; *No tuning* refers to lack of tunable parameters that affect the rendered image or gradients. *GI* denotes global illumination, support for physically-based illumination and shadowing including indirect effects. See text for detailed analysis.

extent, without sacrificing their desirable properties such as correct outputs, a high degree of user control through programmable shading and geometry processing, massive parallelization in all operations, and the ability to render high-resolution images of scenes consisting of millions of geometric primitives.

Concretely, we describe a differentiable rendering system based on deferred shading [Deering et al. 1988], and identify four primitive operations for which we provide custom, high-performance implementations: rasterization, attribute interpolation, texture filtering, and antialiasing. Our modular primitives enable rendering high-resolution images of complex scenes, using arbitrary user-specified shading, directly inside automatic differentiation (AD) frameworks such as TensorFlow or PyTorch.

As a motivating example, we cast facial performance capture as an inverse rendering problem and show that it can be efficiently solved using direct photometric optimization of shape and surface texture in megapixel resolutions. While our proof-of-concept solution does not aim to reconstruct the mouth, eyes, or complex material appearance, the high accuracy of the results in comparison to a state-of-the-art commercial solution demonstrates the viability of high-performance differentiable rendering in solving this problem.

Our differential rasterization primitives are publicly available at https://github.com/NVlabs/nvdiffrast.

2 RELATED WORK

There is a large body of work on using rendering as part of an optimization process that infers properties of the world from images. These include inverse rendering algorithms that fit 3D and appearance models to photographs in an analysis-by-synthesis loop [Patow and Pueyo 2003], as well as techniques that use a 3D renderer as part of a more complex machine learning model.

By far the most common approach in previous work is to design a special-purpose differentiable image synthesis pipeline focusing on the particular requirements of the downstream task, with no particular emphasis on flexibility or generality [Chen et al. 2019; Kato et al. 2018; Liu et al. 2019]. A notable exception is OpenDR [Loper and Black 2014] that, despite its limited shading model, explicitly sets out to develop a general-purpose differentiable rendering system.

Research on general-purpose differentiable rendering divides into two categories depending on whether the primary motivation is image quality or performance. Table 1 summarizes various characteristics of the previous methods analyzed below.

Li et al. [2018] introduce differentiable, physically-based rendering using Monte Carlo ray tracing with proper visibility gradients. Light transport is integrated using random sampling, which leads to noise in the images that diminishes with more sampling. Mitsuba 2 [Loubet et al. 2019; Nimier-David et al. 2019] is a versatile rendering framework that can target a wide array of rendering problems. The primary problem in using these renderers as parts of an intensive optimization or learning task is their lack of performance — to not become a bottleneck over millions of iterations, the rendering times in sufficiently high resolutions should be measured in milliseconds, instead of seconds or minutes needed for recursive light transport simulation. These systems can also be configured to compute only primary visibility and local shading, which is sufficient for many applications. This boosts the performance of Li et al. [2018] to ~100ms in a simple scene in 640×480 resolution, which is still orders of magnitude too slow for many applications.

The second category of differentiable rendering aims at higher performance. Primarily targeted at solving tasks such as shape or pose inference [Chen et al. 2019; de La Gorce et al. 2011; Liu et al. 2019; Loper and Black 2014], they render and shade 3D meshes using local shading only. Strong emphasis is placed on obtaining useful visibility gradients to facilitate shape inference via gradient descent.

Soft Rasterizer [Liu et al. 2019] rasterizes each triangle as a probabilistic cloud with a configurable blur radius; these clouds are combined heuristically based on other configurable parameters. This blur makes coverage a continuous function of vertex positions, which is necessary for obtaining visibility gradients. However, the blur also means that opaque surfaces become transparent around edges, leading to an incorrect image. Optimization thus requires tuning these parameters to reach a balance between image correctness and gradient quality. DIB-R [Chen et al. 2019] renders colors without antialiasing and outputs an additional alpha channel that extends outside the covered pixels by a configurable blur radius. The alpha channel can be used for approximating visibility gradients, but only if an alpha mask is available for reference images as well. Also, these gradients are affected by all triangles regardless of occlusion. Because color channels are point sampled, no visibility gradients are obtained for silhouettes that are in front of other geometry. This is insufficient for, e.g., determining hand poses [de La Gorce et al. 2011], and would also fail if one were to render, e.g., a skybox behind the mesh, so the method cannot be considered general-purpose. As with Soft Rasterizer, it is not obvious how the blur parameter should be set. Neural Mesh Rendering [Kato et al. 2018] produces the image using point sampling and no antialiasing, and in the backward pass hallucinates image-based gradients on triangle edges based on the geometry. The gradients are thus not consistent with the rendered image.

In contrast to these approximations, our aim is to differentiate the standard hardware graphics pipeline without altering its image formation principles. This places special emphasis on occlusion by opaque surfaces. For the gradient to be consistent with the forward imaging model, a 3D primitive that has no effect on the image — for instance, due to being off the screen or occluded by other primitives — should, by our premise, receive a zero gradient.

The reparameterization technique of Loubet et al. [2019] used in Mitsuba 2 [Nimier-David et al. 2019] produces correct visibility gradients only when occluders and occludees can be inferred from four samples. Although more complex occlusion scenarios are not handled correctly, Loubet et al. introduce a bandwidth parameter that can be adjusted to reduce the errors at the cost of increased noise. It is argued that handling the most common, simple cases is sufficient for practical purposes, and our approach for visiblity gradients is founded on the same premise. OpenDR [Loper and Black 2014] approximates *all* gradients based on the final image and knowledge of which triangle was rendered into each pixel. This has the unfortunate effect of producing incorrect gradients for effects such as highlights, because all shading is assumed to be "glued" onto the surfaces. As an example, OpenDR's gradients falsely indicate that moving a planar surface tangentially would move the highlights and reflections on it as well. In addition, inferring gradients from the final pixels can be seen as equivalent to taking finite differences instead of analytic gradients. Flexibility is limited because textures can modulate appearance only multiplicatively, and differentiation with respect to textures is not supported.

3 DIFFERENTIABLE RENDERING PRIMITIVES

Given a 3D scene description in the form of geometric shapes, materials, and camera and lighting models, rendering 2D images boils down to two computational problems: figuring out the things that are visible in each pixel, and what color those things appear to be. A proper differentiable renderer has to provide gradients for all the parameters — e.g., lighting and material parameters, as well as the contents of texture maps — used in the process.

For what follows, it is useful to break the rendering process down into the following form, where the final color I_i of the pixel at screen coordinates (x_i, y_i) is given by

$$I_i = \underset{x,y}{\text{filter}}\left(\text{shade}\left(M(P(x,y)), \text{lights}\right)\right)(x_i, y_i). \tag{1}$$

Here, $P(x, y)$ denotes the world point visible at (continuous) screen coordinates (x, y) after projection from 3D to 2D, and $M(P)$ denotes all the spatially-varying factors (texture maps, normal vectors, etc.) that live on the surfaces of the scene. The shade function typically models light-surface interactions. The 2D antialiasing filter, crucial for both image quality and differentiability, is applied to the shading results in continuous (x, y), and the final color is obtained by sampling the result at the pixel center (x_i, y_i). In real-time graphics, these steps are typically approximated by techniques like multisample antialiasing (MSAA).

The geometry, projection, and lights can all be considered as parametric functions. The visible world point is affected by the geometry, parameterized by θ_G, as well as the projection, parameterized by θ_C. Similarly, the surface factors are parameterized by θ_M, and light sources by θ_L.[1] We follow the common view and take differentiable rendering to mean computing the gradients $\partial L(I)/\partial\{\theta_G, \theta_M, \theta_C, \theta_L\}$ of a scalar function $L(I)$ of the rendered image I with respect to the scene parameters. Note that this does not require computing the (very large) Jacobian matrices $[\partial I/\partial\theta_G]$, etc., but rather only the ability to implement multiplication with the Jacobian transpose ("backpropagation"), yielding the final result through the chain rule:

$$\left[\frac{\partial L(I)}{\partial \theta_G}\right] = \left[\frac{\partial I}{\partial \theta_G}\right]\left[\frac{\partial L}{\partial I}\right],$$

and similarly for the other parameter vectors.

Two main factors make the design of efficient rendering algorithms challenging. First, the mapping $P(x, y)$ between world points and screen coordinates is dynamic: it is affected by changes in both scene geometry and the 3D-to-2D projection. Furthermore, it is discontinuous due to occlusion boundaries. These two factors are also central points of difficulty in computing the gradients we seek. The following sections outline our approach to addressing them.

3.1 Design Goals

Our overall aim is to implement an efficient differentiable real-time graphics pipeline, with the following specific design goals:

- G1 **Efficiency**. Support modern graphics pipelines' ability to render, in high resolution, 3D scenes that are complex in terms of geometric detail, occlusion, and appearance.
- G2 **Minimalism**. Easy integration with modern automatic differentiation (AD) frameworks, such as PyTorch and Tensorflow, without duplication of features.
- G3 **Freedom**. Support arbitrary user-specified shading, as well as arbitrary parameterizations of input geometry, without committing to specific forms such as the Phong model or blendshapes.
- G4 **Quality**. Support the texture filtering operations required by shaders that implement complex appearance models, while making no assumptions about the contents of the textures. In addition to quality, this is also important for optimization dynamics.

3.2 System Design

Goal G1 immediately precludes algorithms that do not parallelize over both the geometric primitives and pixels, and those that do not properly account for occlusion of overlapping primitives. In the remaining space, we make the following design choices:

- C1 **Modularity**. We identify four modular primitive operations that implement crucial operations in a graphics pipeline. Each primitive is exposed as a backpropagation-capable operation with a fixed input/output interface to the host AD framework. Much like today's configurable and programmable hardware graphics pipelines, this non-monolithic design enables easy construction of potentially complex custom rendering pipelines.
- C2 **Positions and Textures are Tensors**. Our system takes the input geometry and texture maps in the form of tensors from

[1] In the simplest case, θ_G and θ_M, could describe, say, the vertex coordinates of a triangle mesh of a fixed topology and a diffuse albedo stored at the vertices and interpolated into the interiors of triangles; we use the abstract notation to allow for complex parameterizations fed into the renderer from within a deep learning model.

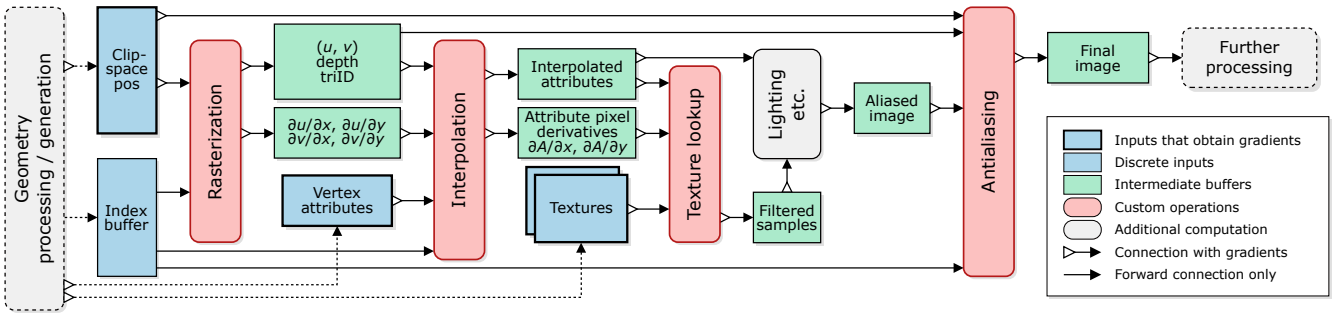


Fig. 1. A simple differentiable rendering pipeline with our proposed primitive operations highlighted in red. The input data for rendering (blue) may be generated by, e.g., a neural network if the pipeline is part of a larger computation graph. In simpler setups the geometry processing might include only the model/view/perspective transformations for vertex positions with other inputs being constants or learnable parameters. All intermediate buffers (green) are in image space. Connections with gradients are denoted by a white triangle. Channel counts are fixed only for vertex positions and indices, and in the intermediate buffers produced by the rasterization operation. There are no restrictions on the channel counts for vertex attributes, textures, related intermediate data, or the output image.

the host AD system. This allows parameterizing both in a freely-chosen manner, and enables our rendering primitives to be used as building blocks of a complex learning system.

C3 **Operate in Clip Space**. Contrary to common differentiable rendering systems, we place it on the user's responsibility to perform world, view, and homogeneous perspective transformations — but not perspective division — on the geometry using the host AD system. By this, we follow the separation between geometry and pixel processing made by all major graphics APIs. We feel this offers the cleanest possible interface between the host AD system and the renderer, further amplifying the benefits of their co-existence.

C4 **Deferred Shading**. We build on the concept of deferred shading [Deering et al. 1988]. This entails first computing, for each pixel, the $M(P(x, y))$ terms from Equation (1) and storing the intermediate results in an image-space regular grid. The grid is subsequently consumed by the shading function. As shading is performed on a regular grid, it can be implemented entirely outside our rendering primitives using the efficient dense tensor operations in the host AD library, in line with G2 and G3.

C5 **Image-space Antialiasing**. We approach differentiation of coverage in image space, approximating the inputs of the antialiasing filter in Equation (1) by the output grid of the deferred shading pass. Effectively, we assume shading to be constant with respect to the coverage effects at silhouette boundaries, but not with respect to other effects in appearance.

C6 **Triangles**. We focus on triangle meshes as the basic geometric primitive, and seek to utilize the modern graphics pipelines' immensely optimized rasterization subsystem to maximal extent.

We build pipelines out of the following four primitive operations customized for gradient computation. Figure 1 illustrates an example graphics pipeline built out of them.

Rasterization implements the dynamic mapping between world coordinates and discrete pixel coordinates. Leveraging the hardware rasterizer, we store per-pixel auxiliary data in the form of barycentric coordinates and triangle IDs in the forward pass. Using barycentrics as a base coordinate system allows easy coupling of shading and interpolation, as well as combining texture gradients with geometry gradients in the backward pass.

Interpolation is a pipeline operation that expands user-defined per-vertex data (i.e., vertex attributes) to pixel space. Making use of the barycentrics computed by the rasterizer, the interpolator module manages this mapping in both directions.

Texture filtering is a key operation in a shading system. Taking as inputs the interpolated texture coordinates and their screen-space derivatives for MIP-mapping, as well as texture data tensors, our texture filtering module performs trilinear MIP-mapping with gradients correctly propagated through both input texture coordinates as well as the contents of the (MIP-mapped) texture maps.

Antialiasing is performed on the output of the deferred shading operation, taking as additional inputs the barycentrics, triangle IDs, and vertex positions and indices.

We now proceed to describe each primitive operation in detail. For simplicity, the following discussion assumes that a single image is being rendered. However, a differentiable renderer is typically used in stochastic gradient descent -type schemes using minibatches of multiple rendered images. All our operations efficiently support minibatching.

3.3 Rasterization

As per widely adopted graphics API standards, our rasterization module consumes triangles with vertex positions given as an array of clip-space homogeneous coordinates (x_c, y_c, z_c, w_c). We leave it as the user's responsibility to compute clip space positions — often, this comprises only a few homogeneous 4×4 matrix multiplications. The backward pass then computes the gradient $\partial L/\partial\{x_c, y_c, z_c, w_c\}$ of the loss L with respect to the clip-space positions, leaving differentiation with respect to any higher-level parameterizations for the host AD library.

Forward pass. In the forward pass, the rasterizer outputs a 2D sample grid, with each position storing a tuple $(ID, u, v, z_c/w_c)$, where ID identifies the triangle covering the sample, (u, v) are barycentric coordinates specifying relative position along the triangle, and z/w corresponds to the depth in normalized device coordinates (NDC).

A special *ID* is reserved for blank pixels. Barycentrics serve as a convenient base domain for interpolation and texture mapping computations further down the pipeline. The NDC depth is utilized only by the subsequent antialiasing module, and does not propagate gradients. As a secondary output, the rasterizer outputs a buffer with the 2×2 Jacobian of the barycentrics w.r.t. the screen coordinates $J_{uv} = \partial\{u,v\}/\partial\{x,y\}$ for each pixel. These are later used for determining the footprint for filtered texture lookups.

Internally, the rasterization is performed through OpenGL, leveraging the hardware graphics pipeline.[2] Using the hardware graphics pipeline ensures that the rasterization is accurate and there are, e.g., no visibility leaks due to precision issues. We also automatically get proper view frustum clipping as performed by the hardware. The output values, including the per-pixel Jacobians between barycentrics and screen coordinates, are calculated using an OpenGL fragment shader.

Both TensorFlow and PyTorch implement GPU tensor operations in CUDA. To bridge them with OpenGL, we use the driver's OpenGL/CUDA interoperability API. The API minimizes data copies, using the same physical memory when possible, and never requires data to leave the GPU memory.

Backward pass. The backward pass receives, for each pixel, the gradient $\partial L/\partial\{u,v\}$ with respect to the barycentrics output by the rasterizer, and computes the gradients $\partial L/\partial\{x_c, y_c, z_c, w_c\}$ for each input vertex. The perspective mapping between barycentrics and clip-space positions is readily differentiated analytically, and the necessary output is obtained through

$$\left[\frac{\partial L}{\partial\{x_c,y_c,z_c,w_c\}}\right] = \left[\frac{\partial L}{\partial\{u,v\}}\right]\left[\frac{\partial\{u,v\}}{\partial\{x_c,y_c,z_c,w_c\}}\right]. \tag{2}$$

The gradients w.r.t. the screen-space derivatives of the barycentrics ($\partial L/\partial J_{uv}$) are taken into account in a similar fashion. The backward pass is implemented as a dense operation over output pixels, using a scatter-add operation to accumulate the gradients from the pixels to the correct vertices based on the triangle IDs. It can thus be trivially parallelized using a CUDA kernel.

3.4 Interpolation

Attribute interpolation is a standard part of the graphics pipeline. Specifically, it entails computing weighted sums of vertex attributes, with the weights given by the barycentrics, thereby creating a mapping between the pixels and the attributes.[3]

Generally, vertex attributes can be used for arbitrary purposes. One of their typical uses, however, is to provide 2D coordinates for texture mapping. Because of this, our interpolator module supports computing, in the forward pass, screen-space derivatives $J_A = \partial A/\partial\{x,y\}$ of all or a subset of attributes for later use in determining texture filter footprints and other purposes.

[2]On compute clusters, we use OpenGL with EGL for displayless hardware-accelerated rendering.

[3]Note that with our rendering primitives, one can supply a different index buffer for attribute interpolation than was used for rasterization. This is convenient when source data comes from a modeler such as Autodesk Maya that associates each vertex with a 3D position and a texture coordinate from separately indexed arrays. If attribute interpolation were bundled with rasterization, such flexibility would not be possible.

Forward pass. Consider a single pixel at (x,y). Denoting a vector of attributes associated with the ith vertex by A_i, the attribute indices of the triangle visible in the pixel (x,y) by $i_{0,1,2}$, and the barycentrics generated by the rasterizer by $u = u(x,y)$ and $v = v(x,y)$, the interpolated vector A is defined as

$$A = u\,A_{i_0} + v\,A_{i_1} + (1-u-v)\,A_{i_2}. \tag{3}$$

Given the rasterizer's outputs (per-pixel triangle IDs and barycentrics), implementation of the forward pass is trivial.

The screen-space derivatives for attributes tagged as requiring them are computed using the barycenter Jacobians output by the rasterizer by $\partial A/\partial\{x,y\} = [\partial\{u,v\}/\partial\{x,y\}][\partial A/\partial\{u,v\}]$, where the last Jacobian is simple to derive from Equation (3).

Backward pass. The inputs to the backward pass are the per-pixel gradients $\partial L/\partial A$ w.r.t. the interpolated attributes, as well as gradients w.r.t. the screen-space derivatives of the attributes. Much like the backward pass of the rasterizer, the gradients w.r.t. the attribute tensor are computed by a scatter-add into the tensor, applying the Jacobians $\partial A/\partial\{A_{i_0,i_1,i_2}\} = \{u, v, 1-u-v\}$ to the per-pixel input gradients. By simple differentiation, the gradients w.r.t. the input barycentrics are given by

$$\left[\frac{\partial L}{\partial u}\right] = \left[A_{i_0} - A_{i_2}\right]^{\mathrm{T}}\left[\frac{\partial L}{\partial A}\right],\quad \left[\frac{\partial L}{\partial v}\right] = \left[A_{i_1} - A_{i_2}\right]^{\mathrm{T}}\left[\frac{\partial L}{\partial A}\right]. \tag{4}$$

In the same vein, the gradients w.r.t. the screen-space derivatives of the input barycentrics $\partial L/\partial J_{uv}$ are computed based on the incoming gradients w.r.t. the screen-space derivatives of the attributes $\partial L/\partial J_A$.

3.5 Texture Mapping

We perform texture mapping using trilinear MIP-mapped texture fetches. In the general case, this entails picking a fractional MIP-map pyramid level (i.e., level-of-detail, LOD) based on the incoming screen-space derivatives of the attributes used as texture coordinates, and performing a trilinear interpolation from the eight texels on the appropriate MIP pyramid levels. Figure 2 illustrates our implementation. We choose the MIP level based on the texture-space length of the major axis of the sample footprint as defined by the screen-space derivatives of the texture coordinates. This is conservative in the sense that grazing angles result in blurring instead of aliasing.

Once a pair of MIP-map levels has been picked, operation of the forward and backward passes closely resemble attribute interpolation: On each level, the four closest texels take the place of the three triangle vertices, and the two sub-texel coordinates that determine exact position within the four-pixel ensemble take the place of the barycentrics. Consequently, our implementation is highly similar, with the forward pass requiring a gather, and the backward pass requiring a scatter-add, with the related Jacobians computed with equal simplicity from the bilinear basis functions and texture contents. As the derivations are highly similar, we omit them for space. Note, however, that gradients are computed also for the texture coordinate attributes, as well as for the screen-space derivatives for the texture coordinates.

MIP-mapped texturing differs from attribute interpolation by its multiscale nature: gradients are accumulated on various levels of the MIP-map pyramid in the backward pass. As all MIP-map levels are

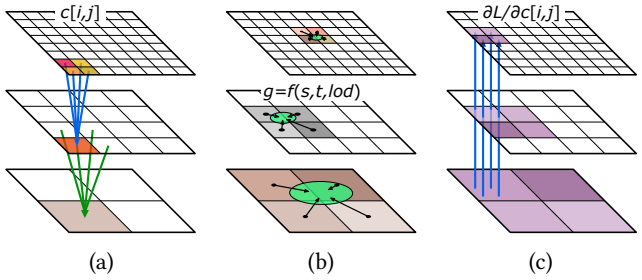


Fig. 2. Filtered differentiable texture lookup with a non-constant texture. (a) In the beginning of forward pass, prefiltered MIP levels c_{lod} are constructed from the full-resolution texture $c[i, j]$ by repeated downsampling using a 2×2 box filter. (b) In forward pass, each lookup $g = f(s, t, lod)$ interpolates prefiltered values on the appropriate MIP level as determined by the size of sample footprint. In backward pass, we receive incoming gradients $\partial L/\partial g$. Texture coordinate gradients $\partial L/\partial s$ and $\partial L/\partial t$ for each lookup are computed based on these and contents of texels that were used in interpolation. Simultaneously, texture image gradients $\partial L/\partial c_{lod}[i, j]$ are accumulated into each MIP level. In a trilinear lookup, these calculations are performed on two adjacent levels and weighted according to the fractional part of lod. (c) To produce outgoing full-resolution texture image gradients $\partial L/\partial c[i, j]$, we sum the accumulated gradients from all MIP levels.

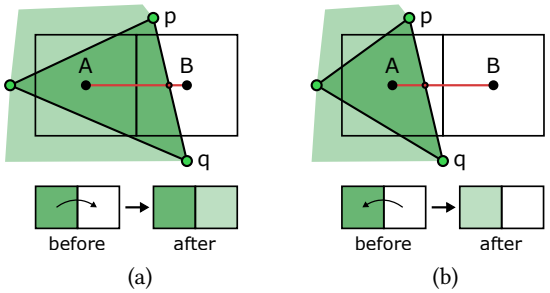


Fig. 3. Illustration of our analytic antialiasing method. A vertical silhouette edge p, q passes between centers of horizontally adjacent pixels A and B. This is detected by the pixels having a different triangle ID rasterized into them. Pixel pair A, B is processed together, and one of the following cases may occur. (a) The edge crosses the segment connecting pixel centers inside pixel B, causing color of A to blend into B. (b) The crossing happens inside pixel A, so blending is done in the opposite direction. To approximate the geometric coverage between surfaces, the blending factor is a linear function of the location of the crossing point — from zero at midpoint to 50% at pixel center. This antialiasing method is differentiable because the resulting pixel colors are continuous functions of positions of p and q.

obtained from the finest-level texture during the construction in the forward pass, the backward pass needs to finish by transposing the construction operation and flattening the gradient pyramid so that the gradient is specified densely at the finest level. Fortunately, this is implemented easily by starting at the coarsest level, recursively up-sampling the result and adding gradients from the next level, precisely like collapsing a Laplacian pyramid.

To the best of our knowledge, no previous differentiable renderer except for Li et al. [2018] has supported differentiable, filtered texture sampling. While we currently do not do so, we note that it would be possible to utilize the hardware texture unit in the forward pass, and retain the CUDA kernel only for the backward pass. The key challenge is that we cannot be certain of the implementation (e.g., numerical precision) of the hardware texture unit, and thus the gradients might not match the forward pass. This will be especially true for anisotropic texture fetches, where considerable freedom exists in covering the footprint [Schilling et al. 1996]. Hardware texture units also require specific memory layouts.

3.6 Analytic Antialiasing for Visibility Gradients

As usual in real-time graphics, we expect shading to be band-limited via filtered texture lookups and other means, and thus not exhibit aliasing within surfaces. However, point-sampled visibility causes aliasing at visibility discontinuities, and more crucially, cannot produce visibility-related gradients for vertex positions. Antialiasing converts the discontinuities to smooth changes, from which the gradients can be computed [Li et al. 2018]. Note that antialiasing can only be performed after shading, and therefore must be implemented as a separate stage instead of bundling it into rasterization.

We follow the same approach as several previous methods [de La Gorce et al. 2011; Loper and Black 2014] and approach the problem by analytic post-process edge antialiasing. Image-based post-process antialiasing is an old and widely-used technique in real-time graphics, with famous techniques such as FXAA being recently superseded by deep learning algorithms [NVIDIA 2018]. For an overview, see Jimenez et al. [2011]. Our method is a variant of distance-to-edge anti-aliasing (DEAA) [Malan 2010] and geometric post-process antialiasing (GPAA) [Persson 2011]. The main differences are in how visibility discontinuities are detected and attributed to vertex positions, as required for computing gradients.

Forward pass. Figure 3 illustrates our antialiasing method. We first detect potential visibility discontinuities by finding all neighboring horizontal and vertical pixel pairs with mismatching triangle IDs. For each potential discontinuity, we fetch the triangle associated with the surface closer to camera, as determined from the NDC depths computed during rasterization. We then examine the edges of the triangle to see if any of them are silhouettes[4] and pass between the neighboring pixel centers. For horizontal pixel pairs, we consider only vertically oriented edges ($|w_{c,1} \cdot y_{c,2} - w_{c,2} \cdot y_{c,1}| > |w_{c,1} \cdot x_{c,2} - w_{c,2} \cdot x_{c,1}|$), and vice versa.

If a silhouette edge crosses the segment between pixel centers, we compute a blend weight by examining where this crossing happens. Pixel colors are then adjusted to reflect the approximated coverage of either surface in the pixels. Essentially this approach approximates the exact surface coverage per pixel [Jalobeanu et al. 2004] using an axis-aligned slab. Consequently the coverage estimate is exact for only perfectly vertical and horizontal edges that extend beyond the pixel. For a diagonal long edge that passes exactly between the pixel centers, the error in coverage is $\frac{1}{8}$th of a pixel.

Finely tessellated surfaces reveal two further approximations. Theoretically, the silhouette between two pixel centers can take

[4]We consider an edge to be on a silhouette if it has only one connecting triangle, or if it connects two triangles that lie on the same side of it after projection. This includes silhouettes that have another surface behind them, unlike DIB-R [Chen et al. 2019] that only considers silhouettes against the background.

any poly-line shape, and the axis-aligned slab approximation can be arbitrarily poor. However, typically additional tessellation manifests itself on the pixel-scale as slightly rounded silhouettes, and for these the approximation accuracy is only slightly worse than for long edges, although exact error bounds cannot be given.

A potentially more serious approximation results from the assumption that all triangles that contain silhouette edges overlap pixel centers and are thus stored during rasterization. Clearly, if we tessellate a surface enough, it is rare for a triangle with a silhouette edge to get rasterized. In this situation, some silhouette edges are not found during antialiasing and no visibility gradient is obtained for these pixels. Occasionally missing a gradient can slow down optimization but rarely prevents it from succeeding, as we show in Section 4.

Backward pass. To prepare for the gradient computation, we store the results of the discontinuity analysis in the forward pass so that we do not have to repeat it in the backward pass. Gradient computation with the stored data is then easy — for each pixel pair that was antialiased in the forward pass, we determine how both vertex positions influence the blending coefficient, and transfer incoming pixel gradients to vertex positions accordingly using scatter-add operations.

3.7 Discussion

Our design aims to do as little as possible apart from managing the complex and dynamic mappings between pixels and the input vertices and textures, leaving the field open for utilizing novel parameterizations for geometry, textures, and lighting models. In particular, the modular design allows the units to be chained many times in a single rendering pipeline. The deferred shading design also leaves many options open. For example, it is easy to perform texture-space shading instead of shading in the pixel domain: after computing shading results using array operations in texture space, one would simply look up the surface colors from the texture in the deferred shading pass.

4 ANALYSIS

In this section, we validate our design principles via targeted, synthetic tests. We first examine the properties of the visibility gradients resulting from our antialiasing, as well as the effects of filtered texture lookups on texture convergence. Then, we construct a rendering pipeline with nontrivial shading and use it to demonstrate an optimization task that involves indirect texture lookups. We also examine a pose fitting task with a difficult optimization landscape. Finally, we measure the performance of our system and compare it to previous differentiable rasterizers.

4.1 Visibility Gradients

To examine the validity of the gradients, we perform a synthetic test where we attempt to infer vertex positions and colors of a simple unit cube. We initialize the solution by taking the true vertex positions and perturbing them randomly in range $[-\frac{1}{2}, \frac{1}{2}]^3$. The vertex colors are initialized to random RGB values in $[0, 1]^3$. We then run Adam optimizer [Kingma and Ba 2015] ($\beta_1 = 0.9$, $\beta_2 = 0.999$) for 5000 iterations, where in each iteration we render the reference mesh and the optimized mesh from same, random viewpoint, and take the image-space L_2 loss between the images. Based on this loss, we learn both vertex positions and colors simultaneously. The learning rate was ramped down exponentially from 10^{-2} to 10^{-4} over the course of the optimization.

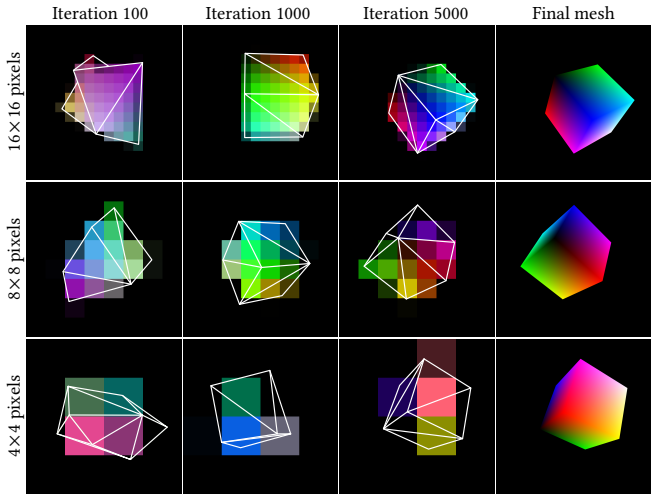


Fig. 4. To validate that our visibility gradients provide useful information even for small triangles, we infer vertex positions and colors of a simple mesh in extremely small resolutions. The geometry of the current solution is superimposed on the rasterized images for illustration purposes only. Rightmost column shows the final, optimized mesh rendered in high resolution. In 4×4 resolution, the average triangle area is only 0.54 pixels. The optimization nonetheless converges to the correct solution, albeit slower than in higher resolutions. In 2×2 resolution the optimization fails to converge.

We implemented two modes for coloring the vertices. In the *continuous* coloring mode, the vertex colors at each corner are shared. This yields a coloring that is continuous across the surface of the cube and has 8 unique colors to optimize. In the *discontinuous* coloring mode, each face has four unique colors at the corners, i.e., a total of 24 unique colors. Consequently, the coloring is not continuous across the edges or vertices of the cube. It can be expected that the latter mode is more difficult to optimize because of the larger number of unknowns and presumably less smooth gradients due to color discontinuities.

Figure 4 illustrates the results in the continuous coloring mode. To our surprise, the optimization succeeds even at 4×4 resolution, where the average size of a rendered triangle is approximately half a pixel. This indicates that our antialiasing-based visibility gradients offer enough information even for small triangles such as those seen in finely tessellated meshes. However, rendering in higher resolution offers faster convergence, highlighting that rendering performance in high resolutions is crucial.

Figure 5 shows the average convergence curves for both coloring modes in the three resolutions tested. Each curve is an average over 10 successful optimization runs. We manually excluded the cases where early optimization steps produced an irrecoverable self-intersecting mesh, which happened in approximately 25% of runs in 4×4 resolution, and less often in higher resolutions. This concurs

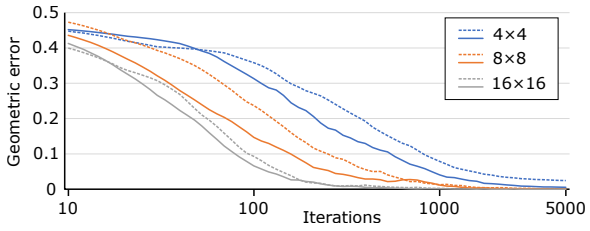


Fig. 5. Convergence of the cube shape and color optimization test (average of 10 successful optimizations). Vertical axis shows the average distance between vertices and their true positions in the unit cube. The solid curves indicate convergence in the continuous coloring mode (Figure 4), and the dashed curves correspond to the discontinuous coloring mode. As expected, the latter is somewhat more difficult to optimize. Note the logarithmic horizontal axis.

with the observation from Mitsuba 2 [Nimier-David et al. 2019] that some rendering-related components require careful initialization. We could have made these configurations less likely by lowering the learning rate, initializing the mesh to further away from self-intersecting states, or by using a suitable regularization term that pushes apart geometry that is in danger of folding over itself. We chose not to use any regularizers in this test because we explicitly wanted our gradients to be based on image-space loss only.

4.2 Texture Filtering

To measure the importance of texture filtering via mipmaps, we constructed a test where we attempt to learn a texture based on synthetic, high-quality reference images that exhibit large variations in scale. We then measure how well the texture is learned with and without mipmapping.

Figure 6a shows example reference images of this task. The reference images are rendered first in 4096×4096 resolution and then downsampled to 512×512 using a high-quality downsampling filter. The reference images are thus well band-limited and display no aliasing, blurring, or other artifacts.

The goal of the optimization is to learn a cube map -parameterized texture with 512×512 pixel faces, mapped onto a unit sphere, based on the reference images. We again use Adam [Kingma and Ba 2015] as the optimizer ($\beta_1 = 0.9, \beta_2 = 0.99$) and run it for 20 000 iterations, ramping the learning rate from 10^{-2} to 10^{-3} during the course of optimization. This learning rate schedule was chosen to be optimal for the case without mipmapping. The training images are rendered directly in 512×512 resolution, from the same, random viewpoints as the reference images, and the optimizer attempts to minimize L_2 loss between training and reference images. The same mesh and texture parameterization are used for all images.

Learning the texture is made difficult by the sphere being placed randomly at distance $[1.5, 50]$ from the camera. Hence some reference images view a close-up patch of the surface, whereas most are too distant to infer texel-level details. This replicates the effects of highly variable pixel-to-texel ratio in reference imagery, which we expect to be present in many kinds of real-world data such as street view images or sets of in-the-wild photographs.

Figures 6b,c illustrate that with mipmapped texture filtering, the learned texture converges to a solution much closer to the reference (32.9 dB vs 25.6 dB). The convergence failure of non-mipmapped version can be explained by a simple thought experiment. When the reference image has a faraway pixel with a large texture footprint, its value is determined by a weighted mean of the reference texture over that footprint. Without mipmapping, we will sample whichever full-resolution texel quad lands under that pixel center. If this value deviates from the large-area average in the reference image, the gradients will pull the texels in the learned texture towards this average. Over many such updates, this pull towards the mean leads to attenuated high-contrast details, which can be seen in Figure 6b where even the converged non-mipmapped solution has less visible contrast than the solution obtained via proper texture filtering.

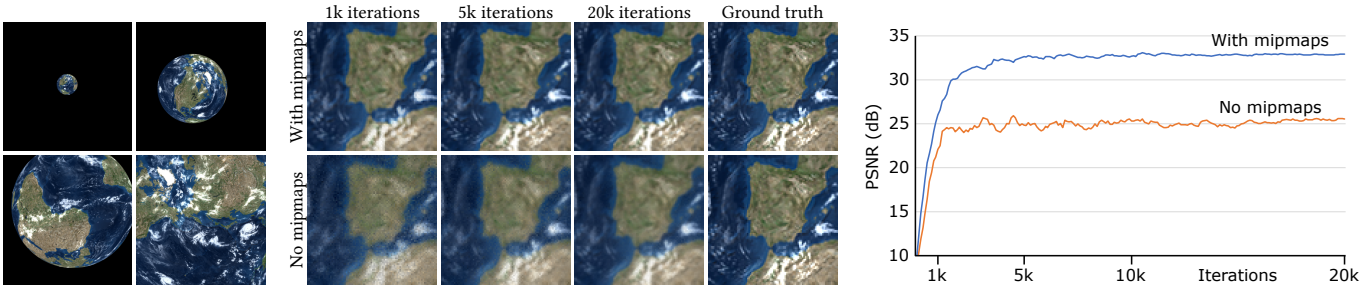
4.3 Indirect Texturing, BRDF Optimization

To demonstrate the flexibility of our modules, we construct a rendering pipeline that computes reflections via environment mapping [Greene 1986] and adds a highlight from an additional light source using a Phong BRDF [Phong 1975]. Figure 7 illustrates the use of this rendering pipeline for solving the environment map contents and Phong BRDF parameters based on the reflections from an irregular object with known geometry and pose.

At the beginning of optimization, the BRDF parameters are initialized to random values, whereas the environment map is initialized to uniform gray. In each iteration, the camera angle and light direction are randomized. Optimization is done using Adam with a fixed learning rate of 10^{-2} and a simple image-space L_2 loss. In this synthetic test, the unknown environment texture and BRDF parameters rapidly converge to the reference solutions.

The rendering pipeline is constructed as follows. We start by rasterizing the geometry as usual, obtaining a frame buffer with per-triangle barycentrics and their screen-space derivatives. We also calculate a normalized reflection vector for each vertex. These reflection vectors are then used as attributes for interpolation, which yields per-pixel reflection vectors and screen-space derivatives for each of their components. We represent the environment map as a cube map, so for each per-pixel reflection vector we determine the corresponding cube map face and 2D texture coordinates within it. The same calculation also yields the screen-space derivatives of the texture coordinates, and we perform a trilinear texture fetch to the environment map. This is important because reflections from curved surfaces introduce highly variable distortions and texture footprint sizes. The cosine between the reflection vector and light direction vector required by the Phong BRDF model is computed based on the per-pixel reflection vectors.

The shading computation involves 18 lines of Python code using standard TensorFlow operations. In our opinion, this is a small price to pay for the complete freedom to tailor shading, data representations, etc., to the needs of the application, compared to incorporating a fixed set of shading models into the differentiable rendering system itself. It would not be possible to implement a similar setup in previous rasterization-based differentiable renderers without modifying their internals.



(a) Example reference images (b) Closeups of learned texture (c) Texture convergence vs. reference

Fig. 6. Filtered texture sampling via mipmaps helps considerably when learning a texture in a difficult geometric setup. See text for full description. (a) Example synthetic reference images in 512×512 resolution. (b) With mipmapping enabled, sampling is prefiltered and gradients are routed to the correct detail levels. Without mipmaps, faraway views yield badly filtered samples and spurious, noisy texture updates that do not converge to the correct solution. (c) Convergence of the learned texture compared to the reference texture. Learning rate schedule was optimized for the "no mipmaps" case, and the same schedule was used with mipmaps.

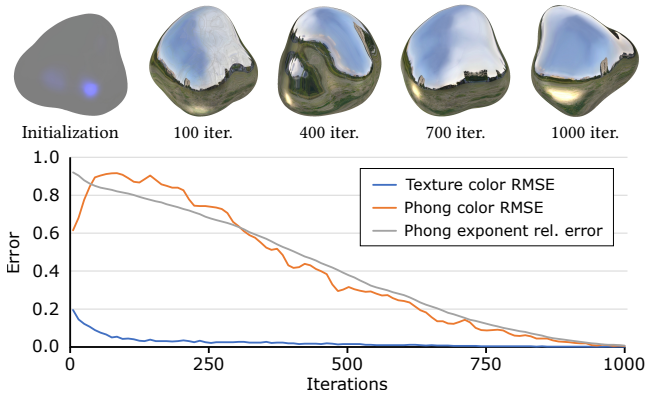


Fig. 7. Optimizing environment map texture and Phong BRDF parameters in a synthetic test case. Top: Example renderings at various iteration counts. The texture converges slightly unevenly due to the distribution of indirect texture lookups, as seen at 100 iterations. Bottom: Convergence of the learned parameters over the course of optimization.

Limitations. Local shading models, such as the one demonstrated here, cannot accurately model global phenomena such as interreflections. If such fidelity is required, a path tracing based differentiable renderer will be necessary [Li et al. 2018; Loubet et al. 2019; Nimier-David et al. 2019]. However, there is no inherent limit on the complexity of the local shading model, so e.g. microfacet [Cook and Torrance 1982] or Gaussian mixture model [Herholz et al. 2016] BRDFs could be used to seek a better fit to data. Ultimately it depends on the intended use how physically accurate the shading should be — even a crude approximation of appearance may be sufficient for inferring other unknowns such as pose or geometry. In Section 5, we demonstrate that in the context of facial performance capture, the per-frame geometry of skin areas can be accurately recovered without any shading at all. In situations like this, striving for physical fidelity would unnecessarily slow down the optimization.

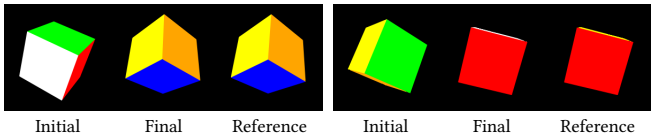


Fig. 8. Two example cases from the cube pose optimization test. With trivial noise-based regularization, we obtain an average error of 48.62° which is an improvement over the 63.57° of Liu et al. [2019], indicating that the blur and transparency offered by SoftRas are not necessary in this task. The average error is dominated by local minima where the pose looks correct but the colors are wrong (see example on the right with ~180° final pose error — the yellow face should be on top instead of white). Customizing the optimization method to suit the task better lowers the average error to 2.61°. Some cases are impossible due to only one face of the cube being visible in the reference image, but they are rare enough to not contribute significantly to the averages.

4.4 Pose Optimization

In the SoftRas paper, Liu et al. [2019] investigate the problem of resolving the pose of a rendered cube using gradient-based optimization of image-space loss. The task is made difficult by an optimization landscape with many local minima (Figure 8). The image synthesis model of SoftRas allows turning all surfaces partially transparent and blurring them by an arbitrary amount. This results in a smoother loss function and a modest improvement in the resolved poses.

To demonstrate that the nonstandard image synthesis model of SoftRas is not necessary for solving this task, we focus on the optimization process instead of manipulating the rendering model. A simple and efficient way to discourage local minima in a stochastic fashion is to add noise to the unknown parameters[5] during optimization. Indeed, running the optimization for 10k iterations using Adam and ramping down the noise strength from 1 to 0.003 over the course of optimization yields an average pose error of 48.62° measured over 100 random trials. This is an improvement over the best result of 63.57° reported by Liu et al. [2019], indicating that

[5]We represent the pose as a quaternion. Noise is applied by constructing a random quaternion and mixing it with the pose using spherical interpolation [Shoemake 1985].

noise-based regularization is at least as effective as their approach based on transparency and blur.

However, we note that gradient descent from a random initial state is an ineffective way to solve this problem. Splitting the optimization into two phases — first greedily seeking for a good initial pose by applying ramped-down noise in a gradient-free fashion, and then continuing with Adam from the pose with the smallest image-space loss — lowers the average error to 22.49°. As a further task-specific optimization, we can take the symmetries of the cube into account and customize the noise to incorporate random symmetry-preserving rotations. This effectively bridges the local minima with similar pose but different color combinations and lowers the average error to 2.61°.

4.5 Performance

To assess the performance of our method, we selected 14 meshes of varying triangle counts from the ShapeNet database [Chang et al. 2015]. We rendered these meshes using both our method and two comparison methods in multiple resolutions. As comparison methods we used the official implementation of Soft Rasterizer [Liu et al. 2019], and PyTorch3D [Ravi et al. 2020], a more recent differentiable rasterization library for PyTorch. The test was set up to include both forward and gradient evaluations, reflecting the total cost of including a rendering operation in an optimization task.

Default γ, σ parameter values were used for Soft Rasterizer. PyTorch3D was set up to render one pixel blur radius, one face per pixel, and soft compositing (SoftGouraudShader). Default bin size heuristic was enabled. We originally intended to include interior scenes in our tests, but Soft Rasterizer could not render "in-scene" viewpoints due to lack of clipping, making triangles behind the camera render erroneously in front of the camera. Therefore, we limited our test to individual objects rendered in front of the camera. All tests were run on a single NVIDIA TITAN V GPU with 12 GB of memory.

Results of the test are summarized in Table 2. We can see that our method is much less sensitive to triangle and pixel counts than the comparison methods. Soft Rasterizer slows down quickly because it tests each triangle for every pixel, and consequently loses to our method by several orders of magnitude with nontrivial triangle counts and resolutions. PyTorch3D fares better than Soft Rasterizer thanks to its coarse-to-fine rasterization architecture. Still, the performance difference is more than an order of magnitude in our favor and grows with high resolutions and triangle counts, highlighting the better scalability of our method.

Occluded vs visible geometry. It is generally desirable that rendering performance is not affected by the amount of geometry that is not visible in the rendered image. Our method employs deferred shading, and is therefore mostly oblivious to occluded or out-of-view geometry except at the rasterization step. The same holds for PyTorch3D when storing just one face per pixel, but its rasterization step is expensive so it is not obvious how hidden geometry affects the overall performance.

To quantify the effects of depth complexity, we constructed pairs of synthetic scenes where the number of triangles and covered pixels are held constant but the depth complexity is varied. Specifically, we repeat a simple base mesh either so that all copies are visible and together cover the image, or so that only one is visible and covers the image while all other copies are hidden behind it. Figure 9 shows the measured performance as function of geometric complexity and geometric setup. Resolution is fixed to 1024×1024 and the settings are otherwise the same as above. PyTorch3D, disregarding the constant cost of ~50 ms, scales strongly with the total area of geometry, occluded or not. This is due to its software rasterizer resolving visibility so late that practically no work is saved if the tested fragment is found to be occluded. The performance of our method is mostly unaffected by the depth complexity, as it uses the hardware rasterizer with efficient hierarchical depth tests. SoftRas scales linearly with geometric complexity regardless of the geometric setup, and does not compare favorably to the other two methods.

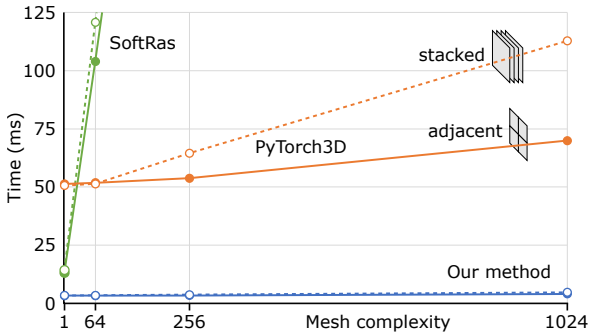


Fig. 9. The effect of geometric configuration on forward + gradient pass execution times in 1024×1024 resolution. A 8×8 grid base mesh (128 triangles) is repeated 1, 64, 256, or 1024 times (horizontal axis). The repeated meshes are either stacked in depth direction (dashed lines) or placed adjacent to each other on the same plane (solid lines). The meshes are scaled so that the output image always has the same number of pixels covered. PyTorch3D [Ravi et al. 2020] scales mostly with the total area of geometry, occluded or not, whereas our method is not slowed down by hidden geometry. SoftRas [Liu et al. 2019] has approximately constant cost per triangle, and cannot keep up with the other two methods.

5 APPLICATION: FACIAL PERFORMANCE CAPTURE

To illustrate the performance and utility of the design of our differentiable rendering pipeline, we examine how to use it to solve markerless facial performance capture, i.e., inferring time-varying facial geometry based on multiple camera streams. This is a non-trivial classical computer graphics problem that has been approached in several ways in the past. Many methods utilize morphable 3D models [Blanz and Vetter 1999] that enable approximating the facial geometry even from monocular data. The downside of this class of methods is that the obtained geometry is approximate and cannot fully reproduce intricate motion. High-quality markerless capture often requires complex capture setups involving structured light or special cameras [Alexander et al. 2009; Bradley et al. 2010]. The passive capture method of Beeler et al. [2011] first reconstructs each frame using a single-shot method [Beeler et al. 2010] and then builds frame-to-frame correspondences iteratively.

While these methods yield great results, they are fairly complex and consequently difficult to implement. As a result, the state of the

Table 2. Performance comparison between our method, the official implementation of Soft Rasterizer (SoftRas) [Liu et al. 2019], and PyTorch3D [Ravi et al. 2020].

	Triangles →	284	352	514	1236	4474	5216	5344	10908	21695	25643	43448	91145	196179	308170
	Resolution	**Rendering + gradients time* (ms)**													
Our method	256×256	2.13	2.03	1.95	1.89	1.95	2.02	2.02	2.08	2.02	2.05	2.00	2.18	2.97	3.12
	512×512	2.26	2.29	2.07	2.08	2.23	2.12	2.08	2.11	2.14	2.20	2.27	2.44	2.66	3.33
	1024×1024	2.70	2.94	2.60	2.56	2.52	2.56	2.61	2.59	2.61	2.64	2.66	3.03	3.36	4.13
	2048×2048	6.21	6.72	4.31	4.95	4.53	4.58	5.31	4.35	4.95	4.46	4.82	5.64	6.46	6.21
	4096×4096	17.73	20.11	12.12	13.18	12.40	12.49	13.29	12.05	12.92	12.39	12.73	14.70	17.73	15.64
SoftRas [2019]	256×256	7.48	6.76	7.84	7.73	18.11	14.29	10.91	29.21	30.93	49.06	65.21	144.57	331.30	788.92
	512×512	10.01	10.42	8.78	10.68	24.05	24.33	24.40	50.09	82.48	99.76	163.51	375.39	865.63	1430.17
	1024×1024	20.73	24.55	15.74	22.00	69.43	73.56	74.07	153.14	277.36	336.54	556.92	1250.74	3012.63	4856.97
	2048×2048	66.25	86.49	46.66	68.43	250.65	276.06	280.30	557.96	1039.49	1234.61	2044.79	4602.11	11487.70	18402.19
	4096×4096	223.88	332.33	163.87	240.71	946.76	1055.14	1082.35	2104.77	4036.47	4768.91	7958.07	17992.60	45499.74	72277.20
PyTorch3D [2020]	256×256	27.12	27.19	26.81	27.95	27.67	27.14	26.94	28.62	27.93	30.08	32.03	37.52	54.82	115.87
	512×512	31.83	30.93	31.08	31.19	31.70	31.82	30.83	32.06	34.84	38.41	41.50	55.53	95.21	158.71
	1024×1024	53.70	53.24	52.03	51.38	52.44	52.17	53.02	53.99	58.98	64.87	83.04	140.34	267.45	438.82
	2048×2048	156.31	153.17	145.34	141.66	144.91	145.48	141.49	148.75	165.77	182.21	259.04	456.07	930.77	1435.20
	4096×4096	571.53	553.69	525.49	513.69	524.58	521.84	508.87	528.99	604.20	677.21	966.76	1754.35	3527.62	5567.26
		Speedup factor													
Our vs SoftRas	256×256	3.51	3.33	4.02	4.09	9.29	7.07	5.40	14.04	15.31	23.93	32.60	66.32	111.55	252.86
	512×512	4.43	4.55	4.24	5.13	10.78	11.48	11.73	23.74	38.54	45.35	72.03	153.85	325.42	429.48
	1024×1024	7.68	8.35	6.05	8.59	27.55	28.73	28.38	59.13	106.27	127.48	209.37	412.79	896.62	1176.02
	2048×2048	10.67	12.87	10.83	13.82	55.33	60.28	52.79	128.27	210.00	276.82	424.23	815.98	1778.28	2963.32
	4096×4096	12.63	16.53	13.52	18.26	76.35	84.48	81.44	174.67	312.42	384.90	625.14	1223.99	2566.26	4621.30
Our vs PyTorch3D	256×256	12.73	13.39	13.75	14.79	14.19	13.44	13.34	13.76	13.83	14.67	16.02	17.21	18.46	37.14
	512×512	14.08	13.51	15.01	15.00	14.22	15.01	14.82	15.19	16.28	17.46	18.28	22.76	35.79	47.66
	1024×1024	19.89	18.11	20.01	20.07	20.81	20.38	20.31	20.85	22.60	24.57	31.22	46.32	79.60	106.25
	2048×2048	25.17	22.79	33.72	28.62	31.99	31.76	26.65	34.20	33.49	40.85	53.74	80.86	144.08	231.11
	4096×4096	32.24	27.53	43.36	38.97	42.30	41.78	38.29	43.90	46.76	54.66	75.94	119.34	198.96	355.96

*Execution times include both forward and gradient evaluations for rendering one frame. Each mesh was rendered several times from multiple angles and the results were averaged to reduce random variation. The exact same meshes, viewpoints, and camera parameters were used for all methods. For our method, we perform rasterization, attribute interpolation, and antialiasing, but no texturing. For Soft Rasterizer, rasterization with default lighting is computed. PyTorch3D was set up to perform Gouraud shading, i.e., attribute interpolation.

art in many cases is using commercial capture systems such as DI4D PRO [2020] or commercial software such as Agisoft Metashape [2020] and R3DS Wrap [2020]. Our goal is not to attempt to surpass this state of the art, but to illustrate how far we can get with a near-trivial formulation as an inverse rendering problem. In particular, we will not attempt to reconstruct tricky regions such as mouth, eyes, or hair, but instead focus on skin areas only.

Our test material consists of three performances captured in a DI4D PRO rig at 29.97 frames per second. There are synchronized 9 camera feeds with 3 in color and 6 in monochrome — for simplicity, we convert the color reference images to monochrome as well prior to processing. Resolution of the reference images is 3008×4112 pixels, and the camera intrinsics and extrinsics are known. Lengths of the three performances ("Neutral", "Disgust", and "Anger") are 89, 123, and 207 frames, respectively.

5.1 Solution via Inverse Rendering

Our goal is to find a global texture and a per-frame mesh so that when rendered from the known camera positions, the textured meshes match the reference footage as closely as possible measured using image-space L_2 loss. We learn the geometry as per-frame deformation of a fixed-topology base mesh that has 16 521 vertices and 16 472 original faces that were triangulated into 32 916 triangles for rendering. The base mesh has texture coordinates referencing a 5:1 aspect ratio texture atlas, and our learned texture is a single-channel 10240×2048 texture initialized to zeros. The texture coordinates of the base mesh are not modified during optimization.

We do not consider material properties or lighting in our image synthesis model, and assume skin to be Lambertian and lighting to be uniform. Neither assumption is valid [Marschner et al. 1999] but we can still expect the geometry to be reconstructed correctly if this produces the best achievable images. Capturing material properties and incident lighting, along with geometry, should be possible with a more complex rendering pipeline [Liu et al. 2017]. Occasional convergence problems were caused by variable shadowing under the nose. We alleviate the problem by passing both rendered and reference images through a high-pass filter so that all low-frequency effects are attenuated before computing the image-space loss. This has the added benefit of making the optimization more resilient to

illumination and shading effects that occur due to changes in head orientation.

5.2 Parameterization and Regularization

In principle, the vertex positions for every frame could be encoded in one large matrix to be optimized. However, we decompose the vertex positions into a matrix product in order to make the optimization landscape more tractable. One could see our model as a three-layer dense neural network that maps a one-hot animation frame index vector into a vector of per-vertex deltas from the base mesh. Even though using a chain of matrices without nonlinearities does not increase the expressiveness of the representation, it accelerates optimization similar to how overparameterization accelerates deep learning [Arora et al. 2018].

Let us denote the vertex count n and the number of reference frames m. Denoting the $3n$ column vector encoding base mesh vertex positions as $\boldsymbol{V}_{base} = [x_1, y_1, z_1, x_2, \cdots]$, the vertex positions for frame i are computed as $\boldsymbol{V}_i = \boldsymbol{V}_{base} + \boldsymbol{M}_3\boldsymbol{M}_2\boldsymbol{M}_1\boldsymbol{w}_i$, where $\boldsymbol{w}_i$ is a one-hot column vector of length m with entry at index i being one. Matrices $\boldsymbol{M}_1$ and $\boldsymbol{M}_2$ are of size $m \times m$, and $\boldsymbol{M}_3$ has size $3n \times m$. The square matrices $\boldsymbol{M}_1$ and $\boldsymbol{M}_2$ are initialized to identity, whereas $\boldsymbol{M}_3$ is initialized to zero. $\boldsymbol{M}_3$ can be seen as a learned basis for the mesh deltas, and $\boldsymbol{M}_1$ and $\boldsymbol{M}_2$ acting as a mapping from frame index to this basis. Finding the geometry thus corresponds to finding values for $\boldsymbol{M}_{1,2,3}$ via optimization.

This formulation has no inherent propensity to, e.g., keep the surface tessellation of the mesh intact. Because of this, we apply a mesh Laplacian regularization term that penalizes local curvature changes compared to the base mesh. Sorkine [2005] gives an overview of Laplacian-based methods for mesh processing. Using their notation, the uniformly-weighted differential $\boldsymbol{\delta}_j$ of vertex $\boldsymbol{v}_j$ is $\boldsymbol{\delta}_j = \boldsymbol{v}_j - \frac{1}{|N_j|}\sum_{k \in N_j} \boldsymbol{v}_k$, where N_j is the set of one-ring neighbors of vertex $\boldsymbol{v}_j$. In other words, $\boldsymbol{\delta}_j$ is the difference between position of vertex $\boldsymbol{v}_j$ and the average position of its neighbors. Our Laplacian regularization term is $L_{\boldsymbol{\delta}} = \frac{1}{n}\sum_{1 \le j \le n} ||\boldsymbol{\delta}_j - \boldsymbol{\delta}_j^{base}||^2$, i.e., the average square Euclidean difference between the vertex differentials of the base mesh and those of the deformed mesh. Although this representation is not rotation-invariant [Lipman et al. 2005], we have not found this to be a problem in practice. Similar regularization has been used in earlier work on shape inference [Liu et al. 2019].

In addition to texture and geometry, we learn global, per-camera brightness and contrast adjustment values that are applied to the rendered images during training. This accounts for differences between reference images originating from color vs monochrome cameras. We do not use any form of temporal regularization, i.e., there are no terms that would prefer nearby frames to be similar to each other. Regardless of this, the solution is temporally stable as can be seen in the accompanying video.

5.3 Optimization

To resolve geometry and texture for a sequence of frames, we initialize the geometry representation as explained above. In each iteration, we choose random frame and camera indices, and render the corresponding mesh using a pipeline similar to one shown in Figure 1. We perform rendering in the same resolution as reference images, i.e., 3008×4112 pixels, which limits our minibatch size to one in practice.

In our primary configuration, optimization is run for 100 000 iterations using Adam optimizer [Kingma and Ba 2015] ($\beta_1 = 0.9, \beta_2 = 0.999$) with a base learning rate of $\lambda = 10^{-3}$. The learning rate is decayed to 10^{-4} during the last 25% of optimization. High-pass filtering is computed as $x' = x - 0.3 \cdot blur(x)$ where x is the rendered/reference image, and $blur(x)$ downsamples the image by a factor of 32×32 and upsamples it back using an approximate Gaussian filter. Images are compared using L_2 loss, so the overall loss function with the Laplacian term included is $L = ||x' - y'||_2^2 + 3L_{\boldsymbol{\delta}}$ where x' and y' are the high-pass filtered rendered and reference images, respectively. A typical optimization run takes 60–70 minutes on a single NVIDIA Tesla V100 GPU, with each optimization iteration taking approximately 40 milliseconds.

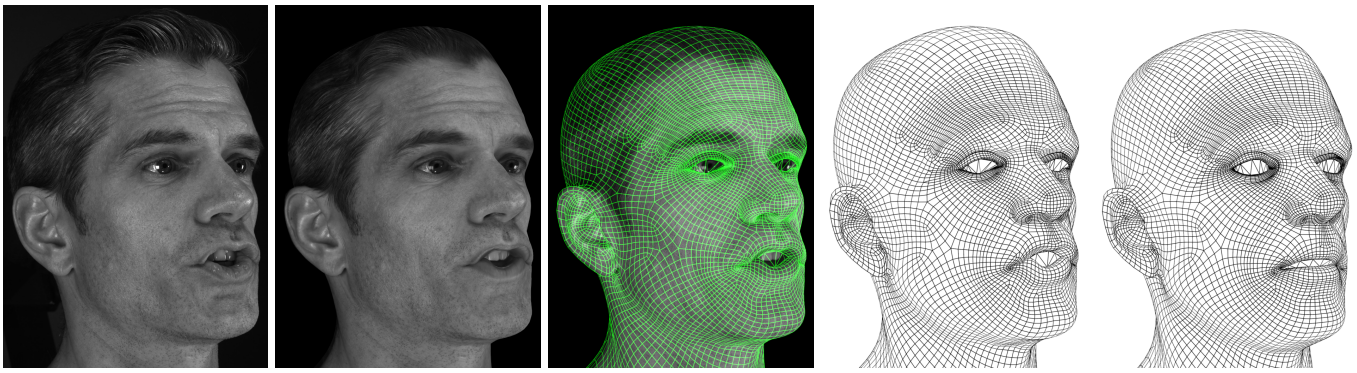
In an additional test, we resolved all three capture sequences in a single optimization run. This task is complicated by the actor being positioned slightly differently in each sequence — with just one base mesh, the alignment is off and large motion of the mesh is required. To circumvent this problem, we also learn a 3×4 rigid transformation matrix $\boldsymbol{R}_j$ for each sequence j, and apply it before the vertex deltas, i.e., $\boldsymbol{V}_i = \boldsymbol{R}_j\boldsymbol{V}_{base} + \boldsymbol{M}_3\boldsymbol{M}_2\boldsymbol{M}_1\boldsymbol{w}_i$. In addition we initialize the texture with one previously solved for the "Neutral" performance, limited $\boldsymbol{M}_2$ to 100×100 elements, and adjusted the dimensions of $\boldsymbol{M}_1$ and $\boldsymbol{M}_3$ accordingly. With these modifications and extending the computation to 800 000 iterations, the optimization successfully found a rigid transformation for each sequence to handle the misalignments, and subsequently solved the vertex deltas for every frame of the combined set along with a texture that best fits all three sequences. As a consequence of optimizing a single texture for the entire material, the mapping between surface and texture-space points becomes automatically consistent between all sequences.

5.4 Results

Figure 10 shows an example result of the geometry and texture optimization in the "Neutral" sequence. For clarity, the wireframes in Figure 10c–e include only the edges of original base mesh instead of the triangulated version. See the accompanying video for the sequences and our reconstructions, as well as the progression of geometry and texture during training.

In our base mesh, the holes for mouth and eyes are simply covered with triangles in order to avoid spurious visibility leaks to the opposite side of the mesh. Obviously, this does not allow faithful reconstruction of mouth and eyes, because eyes have strong view-dependent reflections, and mouth has complex internal geometry. As such, the optimization ends up texturing these covering triangles only somewhat believably. Hair becomes similarly approximated by the learned texture.

Figure 11 shows closeups of the nose region in five frames selected from sequences "Neutral" and "Disgust". There is a surprising amount of fine-grained motion, and our method captures this very accurately as illustrated in the figure. We obtained a 3D reconstruction from DI4D [DI4D 2020] for comparison purposes, and it shows markedly less deformation and does not align properly with the



(a) Reference (b) Our reconstruction (c) With wireframe (d) Wireframe only (e) Base mesh

Fig. 10. An example frame from the reconstruction of a 89-frame sequence. (a) One of the 9 camera images for the frame. (b–d) Our reconstructed texture and geometry, rendered from the same viewpoint using the known camera parameters. (e) Base mesh V_{base} used as the starting point for optimization.

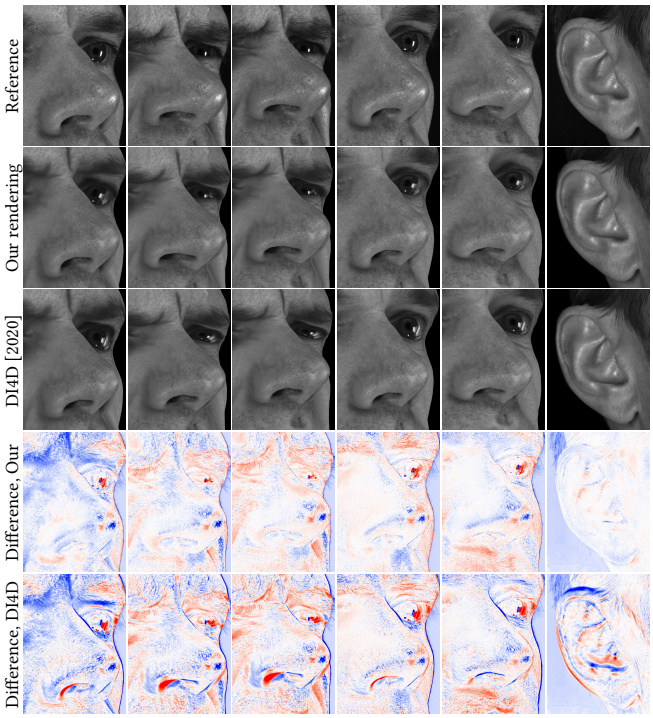


Fig. 11. Closeup of the nose reveals significant motion and deformation. Our solution reproduces the changes in geometry faithfully, while the solution from DI4D [2020] severely attenuates the deformations. False-color difference images (red/blue = brighter/darker than reference) highlight the geometric discrepancies around the nostrils. In our solution, the geometric silhouettes are located correctly and the differences are only due to defocus blur in the reference images that is lacking in our renderings. Right: The DI4D solution did not attempt to recover the motion of ears. We placed no constraints on which vertices are allowed to move during optimization, and thus also captured this motion.

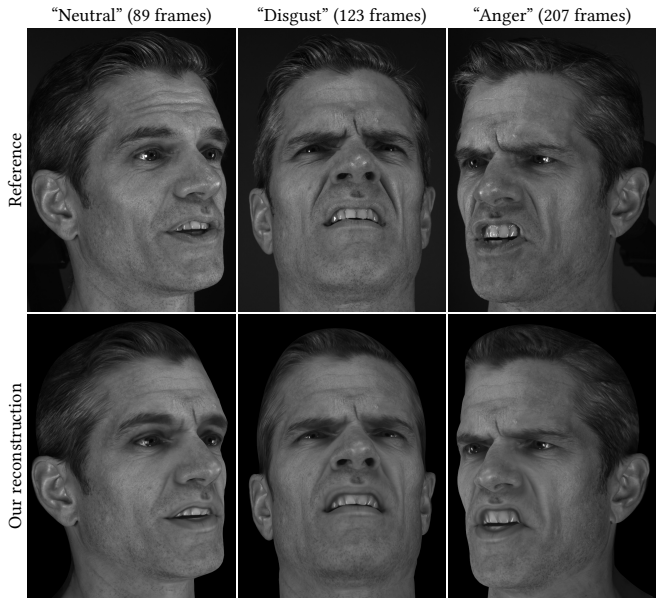


Fig. 12. Example reconstructions from the optimization of three sequences as a single 419-frame sequence, starting from the same base mesh as shown in Figure 10e. Because a single texture is solved for all frames, the vertex to skin correspondence is consistent across all sequences.

camera images. Their optical flow based reconstruction does not attempt to track areas that lack high-quality multi-view observations such as ears and nostrils, whereas our solution automatically reconstructs this motion as well.

Figure 12 shows three example frames from the test where all three performances were resolved at once. In these selected frames, the mouth region looks reasonable, but the "Anger" sequence exhibits artifacts around the mouth in many frames. This is not surprising — our model is unable to render the mouth adequately, so the optimum may be far from correct. Nonetheless, the solution is otherwise temporally stable, and the motion and texture of skin

areas are reconstructed well. As the same texture and texture coordinates are used for all frames, our method provides highly consistent vertex to skin correspondence across all sequences.

6 DISCUSSION AND FUTURE WORK

We have demonstrated a modular differentiable renderer design capable of rendering high-resolution images of complex 3D scenes up to several orders of magnitude faster than prior approaches, while supporting crucial features such as filtered texture mapping with correct gradients. We believe that a high-performance differentiable renderer enables countless uses in inverse graphics, generative modeling, and other computer vision and AI problems, and to help this development, have made our library publicly available at https://github.com/NVlabs/nvdiffrast.

As a practical use case whose success hinges on high-performance differentiable rendering performance, we have demonstrated that multi-view facial performance capture from synchronized high-resolution video cameras can be solved accurately by casting it as a simple inverse rendering problem. It will be interesting to extend this solution to joint material appearance capture, dynamic textures, as well as custom solutions for the eyes, mouth, and hair, integrated as a single optimization problem.

There may exist specific circumstances and applications (e.g., discovery of occluded geometry) where the all-transparent image formation model used by several earlier differentiable renderers may be beneficial for optimization. However, we believe that such problems can also be approached in a principled way via careful choices for mesh parameterization, regularization, and optimization methods, while following the standard occlusion model of the modern graphics hardware pipeline.

ACKNOWLEDGMENTS

We thank Simon Yuen for providing input and comparison data for the facial performance capture experiment, David Luebke for comments, and Sanja Fidler and Wenzheng Chen for discussions on previous work.

REFERENCES

Agisoft. 2020. Agisoft Metashape. https://www.agisoft.com/

Oleg Alexander, Mike Rogers, William Lambeth, Matt Chiang, and Paul Debevec. 2009. The Digital Emily Project: Photoreal Facial Modeling and Animation. In *ACM SIGGRAPH 2009 Courses (SIGGRAPH '09)*.

Sanjeev Arora, Nadav Cohen, and Elad Hazan. 2018. On the Optimization of Deep Networks: Implicit Acceleration by Overparameterization. In *ICML (Proceedings of Machine Learning Research, Vol. 80)*. 244–253.

Thabo Beeler, Bernd Bickel, Paul Beardsley, Bob Sumner, and Markus Gross. 2010. High-Quality Single-Shot Capture of Facial Geometry. *ACM Trans. Graph.* 29, 4 (2010).

Thabo Beeler, Fabian Hahn, Derek Bradley, Bernd Bickel, Paul Beardsley, Craig Gotsman, Robert W. Sumner, and Markus Gross. 2011. High-Quality Passive Facial Performance Capture Using Anchor Frames. *ACM Trans. Graph.* 30, 4 (2011).

Volker Blanz and Thomas Vetter. 1999. A Morphable Model for the Synthesis of 3D Faces *(SIGGRAPH '99)*. 187–194.

Derek Bradley, Wolfgang Heidrich, Tiberiu Popa, and Alla Sheffer. 2010. High Resolution Passive Facial Performance Capture. *ACM Trans. Graph.* 29, 4 (2010).

Angel X. Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. 2015. *ShapeNet: An Information-Rich 3D Model Repository.* Technical Report arXiv:1512.03012 [cs.GR]. Stanford University — Princeton University — Toyota Technological Institute at Chicago.

Wenzheng Chen, Jun Gao, Huan Ling, Edward Smith, Jaakko Lehtinen, Alec Jacobson, and Sanja Fidler. 2019. Learning to Predict 3D Objects with an Interpolation-based Differentiable Renderer. In *Advances In Neural Information Processing Systems*.

R. L. Cook and K. E. Torrance. 1982. A Reflectance Model for Computer Graphics. *ACM Trans. Graph.* 1, 1 (1982), 7–24.

Martin de La Gorce, David J. Fleet, and Nikos Paragios. 2011. Model-Based 3D Hand Pose Estimation from Monocular Video. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 9 (2011), 1793–1805.

Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. 1988. The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics. In *SIGGRAPH '88*. 21–30.

DI4D. 2020. DI4D PRO System. https://www.di4d.com/di4d-pro/

Ned Greene. 1986. Environment Mapping and Other Applications of World Projections. *IEEE Computer Graphics and Applications* 6, 11 (1986), 21–29.

Sebastian Herholz, Oskar Elek, Jiří Vorba, Hendrik Lensch, and Jaroslav Křivánek. 2016. Product Importance Sampling for Light Transport Path Guiding. *Computer Graphics Forum* 35, 4 (2016), 67–77.

André Jalobeanu, Frank O. Kuehnel, and John C. Stutz. 2004. Modeling Images of Natural 3D Surfaces: Overview and Potential Applications. *2004 Conference on Computer Vision and Pattern Recognition Workshop* (2004).

Jorge Jimenez, Diego Gutierrez, Jason Yang, Alexander Reshetov, Pete Demoreuille, Tobias Berghoff, Cedric Perthuis, Henry Yu, Morgan McGuire, Timothy Lottes, Hugh Malan, Emil Persson, Dmitry Andreev, and Tiago Sousa. 2011. Filtering Approaches for Real-Time Anti-Aliasing. In *ACM SIGGRAPH Courses*.

Hiroharu Kato, Yoshitaka Ushiku, and Tatsuya Harada. 2018. Neural 3D Mesh Renderer. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *ICLR*.

Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. 2018. Differentiable Monte Carlo Ray Tracing through Edge Sampling. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 37, 6 (2018), 222:1–222:11.

Yaron Lipman, Olga Sorkine, David Levin, and Daniel Cohen-Or. 2005. Linear Rotation-Invariant Coordinates for Meshes. *ACM Trans. Graph.* 24, 3 (2005), 479–487.

Guilin Liu, Duygu Ceylan, Ersin Yumer, Jimei Yang, and Jyh-Ming Lien. 2017. Material Editing Using a Physically Based Rendering Network. *ICCV* (2017), 2280–2288.

Shichen Liu, Tianye Li, Weikai Chen, and Hao Li. 2019. Soft Rasterizer: A Differentiable Renderer for Image-based 3D Reasoning. In *ICCV*.

Matthew M. Loper and Michael J. Black. 2014. OpenDR: An Approximate Differentiable Renderer. In *ECCV 2014*, Vol. 8695. 154–169.

Guillaume Loubet, Nicolas Holzschuch, and Wenzel Jakob. 2019. Reparameterizing Discontinuous Integrands for Differentiable Rendering. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 38, 6 (2019).

Hugh Malan. 2010. Edge Anti-aliasing by Post-Processing. In *GPU Pro*, Wolfgang Engel (Ed.). A K Peters, 265–289.

Steve Marschner, Stephen H. Westin, Eric P. Lafortune, Kenneth E. Torrance, and Donald P. Greenberg. 1999. Image-Based BRDF Measurement Including Human Skin. In *Rendering Techniques*.

Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. 2019. Mitsuba 2: A Retargetable Forward and Inverse Renderer. *ACM Trans. Graph.* 38, 6 (2019).

NVIDIA. 2018. *NVIDIA Turing GPU Architecture.* Technical Report.

Gustavo Patow and Xavier Pueyo. 2003. A Survey of Inverse Rendering Problems. *Computer Graphics Forum* 22, 4 (2003), 663–687.

Emil Persson. 2011. Geometric Post-Process Anti-Aliasing. http://www.humus.name/index.php?page=3D&ID=86

Bui Tuong Phong. 1975. Illumination for Computer Generated Pictures. *Commun. ACM* 18, 6 (1975), 311–317.

R3DS. 2020. R3DS Wrap. https://www.russian3dscanner.com/

Nikhila Ravi, Jeremy Reizenstein, David Novotny, Taylor Gordon, Wan-Yen Lo, Justin Johnson, and Georgia Gkioxari. 2020. PyTorch3D. https://github.com/facebookresearch/pytorch3d

Andreas Schilling, Günter Knittel, and Wolfgang Strasser. 1996. Texram: A Smart Memory for Texturing. *IEEE Computer Graphics and Applications* 16, 3 (1996), 32–41.

Ken Shoemake. 1985. Animating Rotation with Quaternion Curves. *SIGGRAPH Comput. Graph.* 19, 3 (1985), 245–254.

Olga Sorkine. 2005. Laplacian Mesh Processing. In *Eurographics 2005 - State of the Art Reports*.