THOMAS DELIOT, ERIC HEITZ, and LAURENT BELCOUR, Intel Corporation, France



Fig. 1. In-engine optimization of assets in their engine-specific geometry and material representations. Here, we optimize a control mesh of 2K triangles that controls the tessellation of a Catmull-Clark subdivision surface. The surface has 50K triangles after two levels of subdivision, which are further displaced, normal mapped and shaded with 1024² physically based textures. Timings are for an Intel Arc 770 GPU.

We show how to transform a non-differentiable rasterizer into a differentiable one with minimal engineering efforts and no external dependencies (no Pytorch/Tensorflow). We rely on *Stochastic Gradient Estimation*, a technique that consists of rasterizing after randomly perturbing the scene's parameters such that their gradient can be stochastically estimated and descended. This method is simple and robust but does not scale in dimensionality (number of scene parameters). Our insight is that the number of parameters contributing to a given rasterized pixel is bounded. Estimating and averaging gradients on a per-pixel basis hence bounds the dimensionality of the underlying optimization problem and makes the method scalable. Furthermore, it is simple to track per-pixel contributing parameters by rasterizing ID- and UV-buffers, which are trivial additions to a rasterization engine if not already available. With these minor modifications, we obtain an in-engine optimizer for 3D assets with millions of geometry and texture parameters.

$\label{eq:ccs} \text{CCS Concepts:} \bullet \textbf{Computing methodologies} \to \textbf{Rasterization}.$

ACM Reference Format:

Thomas Deliot, Eric Heitz, and Laurent Belcour. 2024. Transforming a Non-Differentiable Rasterizer into a Differentiable One with Stochastic Gradient Estimation. *Proc. ACM Comput. Graph. Interact. Tech.* 7, 1 (May 2024), 15 pages. https://doi.org/10.1145/3651298

Authors' address: Thomas Deliot, thomas.deliot@intel.com; Eric Heitz, eric.heitz@intel.com; Laurent Belcour, Intel Corporation, Grenoble, France, laurent.belcour@intel.com.

^{© 2024} Copyright held by the owner/author(s). Publication rights licensed to ACM.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, https://doi.org/10.1145/3651298.

1 INTRODUCTION

Motivation for differentiable rendering. A differentiable renderer is a rendering engine that computes a 2D image for a given 3D scene and has, in addition, the ability to provide gradients for the 3D scene parameters via backpropagation through the rendering calculations. The benefits of having these gradients is that it makes possible to optimize the 3D scene parameters to obtain a target 2D image via gradient descent. This allows for many applications such as object placement [Rhodin et al. 2015], object reconstruction [Kato and Harada 2019; Wu et al. 2023], model simplification [Hasselgren et al. 2021], material estimation [Azinovic et al. 2019], etc.

Objective. We assume that a rasterization engine is available and we wish to use differentiable rendering to optimize assets for their final in-engine rendering. Ideally, the solution should keep the workflow simple and self-contained, *i.e.* without using other tools and dependencies than the engine itself. In this context, implementing a renderer from scratch within a differentiable frameworks such as Dr.JIT [Jakob et al. 2022] or Slang.D [Bangaru et al. 2023] is not an option. Using existing differentiable rasterizers such as NVDIFFRAST [Laine et al. 2020] requires externalizing the workflow and relying on external (sometimes vendor-specific) dependencies, which is also problematic. This is why we aim at transforming an existing non-differentiable rasterizer into a differentiable one.

Contribution. Our method is based on the concept of *Stochastic Gradient Estimation* [Fu 2005], a stochastic variant of finite differentiation that allows for estimating gradients without a differentiable framework. However, akin to finite differentiation, this method does not scale to high-dimensional problems: the more dimensions, the noisier the gradient estimates, the more optimization steps are required. Our idea is to cut down the dimensionality by estimating gradients on a per-pixel basis rather than on the whole image. Indeed, the number of parameter contributing to a given rasterized pixel is of tractable dimensionality, regardless of the total number of parameters in the scene. This idea yields a method to make an existing rasterizer differentiable. Namely:

- It is **simple to implement**. Our base differential rasterization component consists of adding ID/UV-buffers to the existing raster targets and two compute shaders.
- It **keeps the workflow self-contained** by bringing the benefits of differentiable rasterization to an existing conventional rasterizer without requiring external dependencies.
- It is **cross-platform** since it uses only conventional graphics API functionalities. This is a significant bonus point for adoption given that existing differential rendering solutions are bound to vendor-specific hardware and/or software.
- It is efficient and scales well in scene complexity. We optimize scenes with 1M+ parameters on a customer GPU. Furthermore, despite stochastic differentiation with noisy gradients being theoretically less efficient than backpropagated differentiation with clean gradients, our implementation is qualitatively on par with NVDIFFRAST [Laine et al. 2020] in our experiments. This is because the gain in speed of remaining in an existing and well-optimized rasterization engine, in contrast to switching to a significantly slower Pytorch environment, finally compensates for the slower convergence due to noisier gradients.
- It covers multiple use cases. We estimate gradients for meshes, displacement mapping, Catmull-Clark subdivision surfaces [Catmull and Clark 1978], semi-transparent geometry, physically based materials, 3D volumetric data and 3D Gaussian Splats [Kerbl et al. 2023].
- Our scope is **raster graphics** (direct visibility only). Our method does not cover further rendering events such as shadows or multiple-bounce illumination.

2 PREVIOUS WORK

2.1 Differentiable rasterization

Differentiable rasterization usually revolves around smoothing the discontinuous visibility function to make it differentiable [Kato et al. 2018; Liu et al. 2019; Loper and Black 2014]. The state-of-the-art framework is NVDIFFRAST [Laine et al. 2020], a performant and modular differentiable rasterizer, which we use as a comparison baseline.

Note that all these methods require a Pytorch/Tensorflow context with external dependencies and are often vendor-specific. We position our method as an in-engine, dependency-free and cross-platform alternative. Our experiments on mesh and texture optimization show qualitatively that it is competitive in terms of optimization speed for these applications.

2.2 Stochastic Gradient Estimation

The concept of estimating gradients in a stochastic manner by applying random perturbations to the input comes in many flavors and under many names such as *Stochastic Gradient Estimation* [Fu 2005], *Monte Carlo Gradient Estimation* [Patelli and Pradlwarter 2010], *Gradient Estimation Via Perturbation Analysis* [Glasserman 1991], *Perturbed Optimization* [Berthet et al. 2020], and many others.

We use one of the variants presented in *Stochastic Gradient Estimation* [Fu 2005], a stochastic variant of finite differentiation. We found it to be the simplest one to convey while proving competitive enough in our experiments.

2.3 Differentiable Rendering with Stochastic Gradient Estimation

Variants of stochastic gradient estimation have already been transposed to the field of rendering [Fischer and Ritschel 2023; Le Lidec et al. 2021]. In this context, it consists of randomly perturbing the 3D scene parameters such that the variation of the 2D image error averaged over the perturbations provides an unbiased estimate of the 3D scene parameter gradients. With this, the scene parameters can be optimized to match a target image. The approach of Fischer and Ritschel. [2023] is especially close to ours because it can be directly implemented within an existing renderer without further dependencies. However, these methods do not scale to high-dimensional problems: the more dimensions, the noisier the gradient estimates, the more optimization steps are required. They are thus limited to low-dimensional problems such as 6D pose estimation or optimizing low-poly meshes.

The key difference of our method is that it estimates gradients on a per-pixel basis rather than on the whole image. Thanks to this, it scales up to scenes with 1M+ parameters such as dense or textured meshes.

2.4 Differentiable Monte Carlo Rendering

Differentiable Monte Carlo path tracers that account for illumination effects beyond direct visibility have been developed [Li et al. 2018; Nimier-David et al. 2019; Vicini et al. 2021; Zhang et al. 2020]. They come with dedicated algorithms to cover difficult cases such as silhouettes and shadows [Bangaru et al. 2020; Loubet et al. 2019; Yan et al. 2022].

Our method is solely based on rasterization (direct visibility) and excludes multiple-bounce effects. We hence do not compete with this line of work.

3 BACKGROUND ON STOCHASTIC GRADIENT ESTIMATION

In this section, we provide background on *Stochastic Gradient Estimation*, a stochastic variant of the finite-difference method. We refer the reader to the work of Fu [2005] for more details.

Problem statement. We consider a *d*-dimensional space of parameters $\theta = (\theta_1, ..., \theta_d)$ where *d* is large and an objective function $f(\theta) \in \mathbb{R}^+$. Our objective is to solve the minimization problem:

$$\min_{\boldsymbol{\theta} \in \mathbb{R}^d} \quad f(\boldsymbol{\theta}). \tag{1}$$

For this purpose, we wish to use a gradient-descent optimizer. We thus need a way to evaluate

$$\frac{\partial f}{\partial \theta} = ?$$
 (2)

In machine-learning frameworks (Pytorch/Tensorflow), this gradient is estimated via backpropagation. We wish to find an alternative way to estimate this gradient when a backpropagation machinery is not available.

Finite difference. The classic finite-difference method method computes a numerical derivative in each dimension by perturbing each component with a small offset:

$$\frac{\partial f}{\partial \theta_i} = \frac{f(\theta + \boldsymbol{b}_i \odot \boldsymbol{\epsilon}) - f(\theta - \boldsymbol{b}_i \odot \boldsymbol{\epsilon})}{2 \, \boldsymbol{\epsilon}_i}.$$
(3)

where $\boldsymbol{\epsilon} = (\epsilon_1, .., \epsilon_d)$ is a user-defined perturbation magnitude vector and $b_i = (0..0, 1, 0..0)$ is the *i*-th basis vector. We note $\boldsymbol{b}_i \odot \boldsymbol{\epsilon}$ the element-wise product of both vectors. The limitation of this approach is that it requires two evaluations of f() per dimension, which makes it untractable in high-dimensional spaces.

Stochastic finite difference. To overcome the dimensionality problem of the finite-difference method, a variant consists of randomly perturbing all the dimensions simultaneously to obtain a stochastic estimator of the gradient:

$$\widehat{\frac{\partial f}{\partial \theta_i}} = \frac{f(\theta + \mathbf{s} \odot \boldsymbol{\epsilon}) - f(\theta - \mathbf{s} \odot \boldsymbol{\epsilon})}{2 \, s_i \, \epsilon_i}.$$
(4)

where $s = (s_1, .., s_d)$ is a random sign vector that contains independent variables $s_i \in \{-1, +1\}$ where each sign has probability $\frac{1}{2}$. The advantage of this method is that two evaluations of f()yield an estimation of the gradient regardless of the number of dimensions. The downside is that the estimator is stochastic, i.e. it is a random variate that is correct on expectation¹ but exhibits some variance. Furthermore, the more dimensions, the higher the variance of the estimator is. In summary, replacing Equation (3) by Equation (4) means trading accuracy for performance.

4 DIFFERENTIABLE RASTERIZATION WITH STOCHASTIC GRADIENT ESTIMATION

We now apply *Stochastic Gradient Estimation* to differential rasterization, where the objective is to optimize a 3D scene such that a rasterized 2D image produced with this scene matches a target image. To do this, we need to estimate the gradients of the rasterization computations.

¹Note that finite-difference methods are biased. The expectation of Equation (4) is thus an approximation of the exact gradient, depending on the perturbation magnitude ϵ . We explain how to set this parameter in practice in Section 4.

4.1 Notations

In this context, the vector $\theta \in \mathbb{R}^d$ represents a 3D scene defined by a set of parameters, typically geometry, textures, etc. A rasterizer computes a 2D image $I(\theta)$ using this 3D scene. Finally, the objective function $f(\theta) = ||I(\theta) - I||^2$ is the error between the rasterized image $I(\theta)$ and a target image I. We summarize these notations in Table 1.

| Name | Domain | Description |
|---|----------------------------------|--|
| $\boldsymbol{\theta} = (\theta_1,, \theta_d)$ | \mathbb{R}^{d} | 3D scene parameters (geometry, textures, etc.) |
| $\boldsymbol{s} = (s_1,, s_d)$ | $\{-1,+1\}^d$ | random sign vector |
| $\boldsymbol{\epsilon} = (\epsilon_1,, \epsilon_d)$ | \mathbb{R}^{d} | perturbation magnitude vector |
| Ι | $\mathbb{R}^{3 	imes W 	imes H}$ | 2D target RGB image |
| $I(\boldsymbol{\theta})$ | $\mathbb{R}^{3 	imes W 	imes H}$ | 2D rasterized RGB image |
| $I_{w,h}$ | \mathbb{R}^3 | pixel (w, h) in target image I |
| $I_{w,h}(\boldsymbol{\theta})$ | \mathbb{R}^3 | pixel (w, h) in rasterized image $I(\theta)$ |
| $f(\boldsymbol{\theta}) = \ I(\boldsymbol{\theta}) - I\ ^2$ | \mathbb{R}^+ | l_2 image error |
| $f_{w,h}(\boldsymbol{\theta}) = \ I_{w,h}(\boldsymbol{\theta}) - I_{w,h}\ ^2$ | \mathbb{R}^+ | l_2 pixel (w, h) error |

Table 1. Notations.

4.2 Per-Pixel Formulation

Motivation. As explained previously, the downside of Equation (4), is that the stochastic gradient estimate is noisy, especially in a high-dimensional parameter space. Intuitively, in our rasterization use case, a large part of this noise can be explained by the fact that the error over the whole image (the error in every pixel) contributes to all the scene parameters. Even if a parameter is never used to compute a pixel it receives noisy gradients from this pixel. In theory, this is not a problem because the noisy gradients conveyed by a pixel not impacted by a parameter are null on expectation. However, in practice, the noisy gradients dramatically burden the gradient decent. In Section 6, we show that this method can hardly be used as is to optimize large scenes. We thus propose a per-pixel gradient computation approach that alleviates this problem and makes the method usable.

Derivation. The error we use is the l_2 error, which is sum of per-pixel errors:

$$f(\boldsymbol{\theta}) = \sum_{(w,h)\in W\times H} f_{w,h}(\boldsymbol{\theta}),\tag{5}$$

and the gradient can be defined in the same way:

$$\frac{\partial f}{\partial \theta_i} = \sum_{(w,h) \in W \times H} \frac{\partial f_{w,h}}{\partial \theta_i}.$$
(6)

Note that if the parameter θ_i is not implicated in the computation of pixel (w, h) then $\frac{\partial f_{w,h}}{\partial \theta_i} = 0$. We can thus rewrite the gradient with a sparse sum where only impacted pixels contribute:

$$\frac{\partial f}{\partial \theta_i} = \sum_{(w,h) \text{ impacted by } \theta_i} \frac{\partial f_{w,h}}{\partial \theta_i}.$$
(7)

By applying the estimator of Equation (4) to Equation (7) we obtain the stochastic gradient estimate our method is based on:

$$\frac{\partial f}{\partial \theta_i} = \sum_{(w,h) \text{ impacted by } \theta_i} \frac{f_{w,h}(\theta + s \odot \epsilon) - f_{w,h}(\theta - s \odot \epsilon)}{2 s_i \epsilon_i}.$$
(8)

In Section 4.3, we show how to implement this equation with a rasterizer and compute shaders.

4.3 Overview

Our differentiable rasterizer implements Equation (8) with 2 compute shaders **P** (perturbation) and **G** (gradient) in addition to the rasterizer \mathcal{R} . We provide an overview of our pipeline in Figure 2.



Fig. 2. **Overview of our differentiable rasterizer.** The first compute shader (**P**) perturbs the scene parameters before they are rasterized (\mathcal{R}). The second compute shader (**G**) accumulates the error differences, which provide a gradient estimate. The key point of our approach is that it accumulates the contribution of a pixel (in red in the images) only in its contributing parameters (in red in the vectors).

| Algorithm 1 Compute shader P (perturbation) | |
|--|--|
| Require: thread ID <i>i</i> | |
| load θ_i, ϵ_i | ⊳ load 2 float |
| $s_i = randomsign()$ | ▶ hash function [Jarzynski and Olano 2020] |
| store $s_i \epsilon_i$, $\theta_i + s_i \epsilon_i$, $\theta_i - s_i \epsilon_i$ | ⊳ store 3 float |
| Algorithm 2 Compute shader G (gradient) | |
| Require: thread ID (w, h) | |
| load $I_{w,h}(\theta + s \odot \epsilon), I_{w,h}(\theta - s \odot \epsilon), I_{w,h}$ | ⊳ load 3 float3 (3× rgb) |
| $f_{w,h}(\boldsymbol{\theta} + \boldsymbol{s} \odot \boldsymbol{\epsilon}) = \left\ I_{w,h} - I_{w,h}(\boldsymbol{\theta} + \boldsymbol{s} \odot \boldsymbol{\epsilon}) \right\ ^2$ | |
| $f_{w,h}(\boldsymbol{\theta} - \boldsymbol{s} \odot \boldsymbol{\epsilon}) = \left\ I_{w,h} - I_{w,h}(\boldsymbol{\theta} - \boldsymbol{s} \odot \boldsymbol{\epsilon}) \right\ ^2$ | |

| for each parameter θ_i contributing to pixel (w, h) do | Implementation of Equation (8) |
|---|--|
| load $s_i \epsilon_i$ | ⊳ load 1 float |
| $\mathbf{AtomicAdd}\left(\frac{\partial f}{\partial \theta_i} \leftarrow \frac{f_{w,h}(\theta + s \odot \epsilon) - f_{w,h}(\theta - s \odot \epsilon)}{2 s_i \epsilon_i}\right)$ | ⊳ atomic add 1 float |
| end for | |
| | |

User-defined perturbation magnitude vector $\boldsymbol{\epsilon}$. Our general methodology to set the perturbation magnitude vector $\boldsymbol{\epsilon}$ is that the perturbation should produce a small but measurable change in the rasterized image. If θ_i is a triangle vertex coordinate, we set ϵ_i such that it results in a perturbation of 1-2 pixels on average in screen-space. If θ_i is a texel (or voxel) parameter, we set ϵ_i to the quantization of the texture (or volume) data format.

Compute shader **P** (*perturbation*). We launch this compute shader over *d* threads (the number of scene parameters) that execute Algorithm 1. The shader computes the perturbed scene parameters $\theta + s \odot \epsilon$ and $\theta - s \odot \epsilon$. Its main ingredient is the generation of the random sign vector *s* via **randomsign**(), which we implement with a random hash function [Jarzynski and Olano 2020].

Rasterization \mathcal{R} . We rasterize the scenes of parameters $\theta + s \odot \epsilon$ and $\theta - s \odot \epsilon$ and obtain two images $I(\theta + s \odot \epsilon)$ and $I(\theta - s \odot \epsilon)$.

Compute shader G (gradient). We launch this compute shader over $W \times H$ threads (the number of pixels) that execute Algorithm 2. The shader computes the pixel errors $f_{w,h}(\theta + s \odot \epsilon)$ and $f_{w,h}(\theta - s \odot \epsilon)$ between the perturbed-scene images $I(\theta + s \odot \epsilon)$ and $I(\theta - s \odot \epsilon)$ and the target image I. Once these errors are available, they provide the gradient estimate for each parameter i contributing to pixel (w, h) following Equation (8). We add the result to the gradient estimate using an **AtomicAdd** operation to avoid interferences between multiple threads (pixels) adding simultaneously their gradient contribution to the same parameter. Note that the critical point of this algorithm is the ability to loop over each parameter i contributing to pixel (w, h). We explain how we achieve this in practice for each type of primitive in Section 4.4.

4.4 Primitives Implementation

Our method uses different strategies depending on the type of content being optimized. For each kind of primitive, we explain how to implement the loop in compute shader **G** (Algorithm 2) over the parameters θ_i contributing to a given pixel (*w*, *h*).

Opaque geometry. We represent opaque geometry with triangles meshes defined by a vertex buffer that stores the 3D vertices and an index buffer that stores the vertices of each triangle. We modify the rasterization pass \mathcal{R} such that, in addition to the RGB output, it rasterizes an ID buffer that contains the index of the rasterized triangle in each pixel. In the compute shader **G**, we sample the ID buffer for each pixel (*w*, *h*) to identify the triangle seen by this pixel and use the index buffer to recover the vertices of this triangle.

Transparent geometry. In the case of transparent geometry, we further modify our rasterization pass \mathcal{R} to support transparent front-to-back rendering with a pre-sorting pass, and output a deep ID buffer with multiple triangle IDs per pixel. This gives us an ordered list of the triangles seen by a pixel. We go through this list in compute shader **G** and proceed in a similar manner as described above for each triangle in the list.

Textures. To optimize texture content, we further modify the rasterization pass \mathcal{R} to rasterize a UV buffer in addition to the RGB output and the ID buffer. It contains the UV coordinates used to fetch the texture in each pixel. In the compute shader **G**, we use these UV coordindates to recover the texel that contributed to the pixel. Note that, in theory, a pixel should contribute to the gradient estimates of all the texels that fall within its texture-space elliptical footprint. In practice, we find that doing so only for the texel closest to the center of the footprint is sufficient if the rendering resolution is high enough to avoid sub-pixel scale texels.

Volumes. To render volumetric content, we ray-march a 3D texture during the rasterization pass \mathcal{R} . In the compute shader **G**, we implement the loop as another pass of ray-marching where each encountered voxel receives gradient update.

5 APPLICATION TO 3D SCENE OPTIMIZATION

We explain how to use the differential rasterizer described in Section 4 to optimize 3D scenes.

Gradient accumulation loop. The differential rasterizer introduced in Section 4 evaluates Equation (8) to obtain a stochastic (noisy) estimate of the gradient. The noisiness of these gradients can burden the gradient descent. It is possible to obtain a lower-variance estimator by averaging N stochastic gradient estimates:

$$\widehat{\frac{\partial f}{\partial \theta_i}} = \frac{1}{N} \sum_{n=1}^N \sum_{(w,h) \text{ impacted by } \theta_i} \frac{f_{w,h}(\theta + s^{(n)}\epsilon) - f_{w,h}(\theta - s^{(n)}\epsilon)}{2 s_i^{(n)} \epsilon_i}.$$
(9)

where the *n*th estimation uses a different random sign vector $s^{(n)}$. We implement this as a loop that repeats *N* times the steps **P**, *G*, and **G**. Note this averaging loop is usually necessary anyways even with deterministic differential rasterizers because there are other sources of noise in the gradients such as the random choice of the point of view. In our case, we randomize our sign vector $s^{(n)}$ simultaneously with these other random variables in each iteration *n*.

Gradient-descent optimizer. After the gradient accumulation loop, we use the gradient estimate to make a gradient descent over the parameters θ . Since our gradient estimate is stochastic, it is preferable to use a gradient-descent optimizer specifically designed for performing stochastic gradient descent such as Adam [Kingma and Ba 2015]. We implement Adam in a compute shader launched over *d* threads (the number of scene parameters) that takes θ and $\frac{\partial f}{\partial \theta}$ as inputs and updates θ . We use it with its default parameters $\beta_1 = 0.9$ and $\beta_2 = 0.999$ and we set the learning rate of each parameter θ_i to the same value as its perturbation amplitude ϵ_i in all our experiments. Note that Adam is invariant to constant scaling factors. In our implementation, we do not perform the division by the constants 2, ϵ_i and N in the denominators of Equation (9).

Additional non-gradient-based optimizations. A gradient descent remains a local exploration of the optimization landscape. In some cases, even with good gradient estimates, the gradient-descent optimizer might be stuck in local minima. Some applications require additional non-gradient-based optimization to converge successfully. For instance, the triangles in Figure 3 or the 3D Gaussian splats in Figure 7 need to be regularly tested and resampled if they become degenerate. We implement this as compute shader launched over the target parameters after each gradient descent. We do not explore thorougly these complementary non-gradient-based optimizations since they are orthogonal to the gradient estimation, which is our core contribution.

6 RESULTS

6.1 Validation of the Per-Pixel Formulation

In Section 4.2, we argue that using the stochastic gradient estimator of Equation (4) as is, with a fullimage error f(), would not converge in high dimensions. This motivates our per-pixel formulation of Equation (8) that we expect to alleviate the dimensionality problem. We test this hypothesis in Figure 3 where we compare the *full-image* approach of Equation (4) and the *per-pixel* approach of Equation (8). In this experiment, each triangle is represented by 12 parameters (3 vertices + 1 RGB color). The three comparisons use respectively 12288 (1K triangles), 122880 (10K triangles), and 1228800 (100K triangles) parameters. Note that the *full-image* approach is conceptually similar to the one of Fischer and Ritschel. [2023], that also estimates the gradient via the impact of perturbations over a full-image error, the only difference being the distribution of perturbation. As expected, optimizing with the *full-image* error is slower and impractical with large numbers of parameters. In contrast, our *per-pixel* variant scales well up to 1M+ parameters.

6.2 Qualitative Comparison to nvDiffRast

Our comparison baseline is NVDIFFRAST [Laine et al. 2020], the state-of-the-art differentiable rasterizer. Note, however, that our objective is not to compete with it in terms of performance or quality. The promise of our method is to provide a simple-to-implement, cross-platform, and dependency-free alternative that can be incorporated into an existing rasterization engine. Still, it is interesting to investigate how both methods compare. To do that, we reproduced two NVDIFFRAST samples provided by Hasselgren et al. [2021] with our implementation. We show these experiments in Figures 4 and 5 and we provide a performance comparison in Table 2. Note that brute force performance is not a relevant measure because both methods behave differently. Indeed, NVDIFFRAST is **slower** because of its Pytorch environment but it provides **clean gradients** that allow for an efficient gradient descent. In contrast, our method executes **faster** within a rasterization engine but provides **noisy gradients**, which make the gradient descent less efficient. We found out that both effects counterbalance each other and that both approaches tend to produce qualitatively similar results with the same amount of optimization time. These experiments hence confirm that our method can be considered as an alternative to NVDIFFRAST for these applications without suffering critical performance or quality penalty.

| | Figures 4 | : | Figures 5 | |
|---|---|------|------------|------|
| image resolution ($W \times H$) | image resolution ($W \times H$) 1024 ² | | 1024^{2} | |
| number of vertices | 1748 | | 1748 | |
| texture resolution | 1024^{2} | | 512^{2} | |
| total number of parameters (<i>d</i>) | 3150972 | | 791676 | |
| | NVDIFFRAST | ours | NVDIFFRAST | ours |
| image/step (N) | 8 | 16 | 8 | 32 |
| time/step | 33ms | 21ms | 33ms | 18ms |

Table 2. Performance comparison on an NVIDIA 4090 GPU.

6.3 Supported Applications

Triangles, textures and volumes. Figures 3, 4 and 5 showcase optimizing triangle soups, meshes, textures and volumes. They are straightforward applications of the implementation described in Section 4.

Subdivision surfaces. In Figure 6, we apply our method to a Catmull-Clark subdivision surface [Catmull and Clark 1978] tessellated on the fly. We optimize the coarse control mesh and the displacement and normal maps that control the final appearance. To support this application, we need our compute shader **G** to associate each tessellated triangle to its original triangle and loop over its neighbors in the control mesh. The subdivision data structure that we use provides a way to do this efficiently [Dupuy and Vanhoey 2021].

Physically based shading. Figure 1 showcases a subdivision surface (same algorithm as the one of Figure 6) with physically based shading using roughness, metallicity, albedo, height and normal maps.

3D Gaussian splats. Figure 7 shows an optimization of 3D Gaussian Splats [Kerbl et al. 2023]. Estimating the gradient is a straightforward application of our transparent geometry support explained in Section 4.4 since the splats are rasterized transparent billboard with a vertex shader and a fragment shader for the shape, color and transparency. To improve the results, we implement an additional resampling and a splat subdivision compute shader executed after each gradient descent, following Kerb et al. [2023].





Fig. 3. Validation of the per-pixel formulation. In this experiment, we optimize triangles soups to match a 2D image. The full-image variant implements Equation (4) where the error over the whole image contributes to every parameter and the per-pixel approach implements Equation (8). The timings are provided for an NVIDIA 4090 GPU.



Fig. 4. **Qualitative comparison against NVDIFFRAST: optimizing a mesh with an albedo map.** We optimize a mesh with 3072 triangles (1748 vertices) and a 1024² albedo texture. The timings are provided for an NVIDIA 4090 GPU.



Fig. 5. Qualitative comparison against NVDIFFRAST: optimizing a mesh with a normal map. We optimize a mesh with 3072 triangles (1748 vertices) and a 512² normalmap texture. The timings are provided for an NVIDIA 4090 GPU.



Fig. 6. **Optimizing a subdivision surface with displacement and normal textures.** We optimize a control mesh of 1K vertices that controls the tessellation of a Catmull-Clark subdivision surface. The surface has 24K triangles after two levels of subdivision, which are further displaced and normal mapped with 1024² textures. The timings are provided for an NVIDIA 4090 GPU.



Fig. 7. **Optimizing 3D Gaussian Splats [Kerbl et al. 2023].** We optimize the splats in a hierarchical manner. The optimizer starts with 1K splats rendered at 128^2 resolution and subdivide them progressively up to 128K splats rendered at resolution 512^2 . The timings are provided for an NVIDIA 4090 GPU.



Fig. 8. **Optimizing a 3D volume.** We optimize a 128³ RGBA volume. The timings are provided for an NVIDIA 4090 GPU.

6.4 Performance Breakdown

In Table 3, we provide more fine-grained performance measures showing the timings for each stage of our method for a single optimization step.

| | Fig. 3 1K triangles | Fig. 3 10K triangles | Fig. 3 100K triangles |
|-------------------|---------------------|----------------------|-----------------------|
| P (×128) | 12µs (×128) | $14 \mu s$ (×128) | 54µs (×128) |
| ${\cal R}$ (×128) | 28µs (×128) | 38µs (×128) | 126µs (×128) |
| G (×128) | 13µs (×128) | 14µs (×128) | 35µs (×128) |
| D | 21µs | 21µs | 105µs |
| Sum | 6.8ms | 8.4ms | 27ms |

Table 3. Performance breakdown for the results of Figure 3. In this experiment, we accumulate N = 128 stochastic gradient estimates before computing a gradient descent step (noted **D** in the table).

7 CONCLUSION

We have proposed a method to transform a non-differentiable rasterizer into a differentiable one. Our experiments have shown that our transformed rasterizer supports the same applications as state-of-the-art differentiable rasterizers without critical performance or qualitative penalty. We successfully used it to optimize triangles, meshes, subdivision surfaces, textures, physically based materials, volumes, and 3D Gaussian splats.

However, we do not position our method as a replacement for other state-of-the-art differentiable rasterizers. Our objective is to bring the benefits of differentiable rasterization to an audience that already possesses a (non-differentiable) rasterization engine and has workflow or platform constraints that prevent using existing differentiable rasterizers. Our method makes it possible to enjoy the possibilities of differentiable rasterization for 3D assets optimization, within the existing engine. We believe that game developers who wish to optimize gaming assets withing their existing workflow will be interested in our method.

REFERENCES

- Dejan Azinovic, Tzu-Mao Li, Anton Kaplanyan, and Matthias Nießner. 2019. Inverse path tracing for joint material and lighting estimation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2447–2456.
- Sai Bangaru, Tzu-Mao Li, and Frédo Durand. 2020. Unbiased Warped-Area Sampling for Differentiable Rendering. ACM Trans. Graph. 39, 6 (2020), 245:1–245:18.
- Sai Bangaru, Lifan Wu, Tzu-Mao Li, Jacob Munkberg, Gilbert Bernstein, Jonathan Ragan-Kelley, Fredo Durand, Aaron Lefohn, and Yong He. 2023. SLANG.D: Fast, Modular and Differentiable Shader Programming. *ACM Transactions on Graphics (SIGGRAPH Asia)* 42, 6 (December 2023), 1–28.
- Quentin Berthet, Mathieu Blondel, Olivier Teboul, Marco Cuturi, Jean-Philippe Vert, and Francis Bach. 2020. Learning with Differentiable Perturbed Optimizers. In Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS'20). Article 797, 12 pages.
- E. Catmull and J. Clark. 1978. Recursively generated B-spline surfaces on arbitrary topological meshes. Computer-Aided Design 10, 6 (1978), 350 – 355.
- J. Dupuy and K. Vanhoey. 2021. A Halfedge Refinement Rule for Parallel Catmull-Clark Subdivision. Computer Graphics Forum 40, 8 (2021), 57–70.
- Michael Fischer and Tobias Ritschel. 2023. Plateau-Reduced Differentiable Path Tracing. In *Proceedings of the IEEE/CVF* Conference on Computer Vision and Pattern Recognition.
- Michael Fu. 2005. Stochastic Gradient Estimation. Technical report (2005).
- Paul Glasserman. 1991. Gradient Estimation Via Perturbation Analysis. Norwell, MA:Kluwer.
- Jon Hasselgren, Jacob Munkberg, Jaakko Lehtinen, Miika Aittala, and Samuli Laine. 2021. Appearance-Driven Automatic 3D Model Simplification.. In *EGSR (DL)*. 85–97.

- Wenzel Jakob, Sébastien Speierer, Nicolas Roussel, and Delio Vicini. 2022. Dr.Jit: A Just-In-Time Compiler for Differentiable Rendering. *Transactions on Graphics (Proceedings of SIGGRAPH)* 41, 4 (2022).
- Mark Jarzynski and Marc Olano. 2020. Hash Functions for GPU Rendering. *Journal of Computer Graphics Techniques (JCGT)* 9, 3 (17 October 2020), 20–38.
- Hiroharu Kato and Tatsuya Harada. 2019. Learning view priors for single-view 3d reconstruction. In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 9778–9787.
- Hiroharu Kato, Yoshitaka Ushiku, and Tatsuya Harada. 2018. Neural 3d mesh renderer. In Proceedings of the IEEE conference on computer vision and pattern recognition. 3907–3916.
- Bernhard Kerbl, Georgios Kopanas, Thomas Leimkuehler, and George Drettakis. 2023. 3D Gaussian Splatting for Real-Time Radiance Field Rendering. ACM Trans. Graph. 42, 4, Article 139 (2023).
- Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization.. In ICLR (Poster).
- Samuli Laine, Janne Hellsten, Tero Karras, Yeongho Seol, Jaakko Lehtinen, and Timo Aila. 2020. Modular Primitives for High-Performance Differentiable Rendering. *ACM Transactions on Graphics* 39, 6 (2020).
- Quentin Le Lidec, Ivan Laptev, Cordelia Schmid, and Justin Carpentier. 2021. Differentiable rendering with perturbed optimizers. Advances in Neural Information Processing Systems 34 (2021).
- 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.
- Shichen Liu, Tianye Li, Weikai Chen, and Hao Li. 2019. Soft rasterizer: A differentiable renderer for image-based 3d reasoning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 7708–7717.
- Matthew M Loper and Michael J Black. 2014. OpenDR: An approximate differentiable renderer. In Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part VII 13. Springer, 154–169.
- Guillaume Loubet, Nicolas Holzschuch, and Wenzel Jakob. 2019. Reparameterizing discontinuous integrands for differentiable rendering. ACM Transactions on Graphics (TOG) 38, 6 (2019), 1–14.
- Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. 2019. Mitsuba 2: A Retargetable Forward and Inverse Renderer. ACM Trans. Graph. 38, 6, Article 203 (nov 2019), 17 pages.
- Edoardo Patelli and Helmut J Pradlwarter. 2010. Monte Carlo gradient estimation in high dimensions. *International journal* for numerical methods in engineering 81, 2 (2010), 172–188.
- Helge Rhodin, Nadia Robertini, Christian Richardt, Hans-Peter Seidel, and Christian Theobalt. 2015. A versatile scene model with differentiable visibility applied to generative pose estimation. In *Proceedings of the IEEE International Conference on Computer Vision*. 765–773.
- Delio Vicini, Sébastien Speierer, and Wenzel Jakob. 2021. Path replay backpropagation: differentiating light paths using constant memory and linear time. ACM Transactions on Graphics (TOG) 40, 4 (2021), 1–14.
- Shangzhe Wu, Christian Rupprecht, and Andrea Vedaldi. 2023. Unsupervised Learning of Probably Symmetric Deformable 3D Objects From Images in the Wild (Invited Paper). *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45, 4 (2023), 5268–5281.
- Kai Yan, Christoph Lassner, Brian Budge, Zhao Dong, and Shuang Zhao. 2022. Efficient estimation of boundary integrals for path-space differentiable rendering. ACM Transactions on Graphics (TOG) 41, 4 (2022), 1–13.
- Cheng Zhang, Bailey Miller, Kai Yan, Ioannis Gkioulekas, and Shuang Zhao. 2020. Path-Space Differentiable Rendering. ACM Trans. Graph. 39, 4 (2020), 143:1–143:19.