Transforming a Non-Differentiable Rasterizer into a Differentiable One with Stochastic Gradient Estimation

Thomas Deliot [1] **Eric Heitz** [1] **Laurent Belcour** [1]

Abstract

We show how to transform a non-differentiable rasterizer into a differentiable one with minimal engineering efforts and no automatic differentiation. To do so, we improve on *Stochastic Gradient Estimation* by using a *Per-Pixel Loss* which leverage the fact that only a few primitives contribute to a given pixel. Estimating gradients on a per-pixel basis bounds the dimensionality of the optimization problem and makes the method scalable. To track parameters contributing to a pixel, we use ID- and UV-buffers, which are often already available or trivial to obtain. With these minor modifications, we obtain an in-engine optimizer for 3D assets with millions of geometry and texture parameters.

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 [1], object reconstruction [2], [3], model simplification [4], material estimation [5], 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 [6] or Slang.D [7] is not an option. Using existing differentiable rasterizers such as NVDIFFRAST [8] 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* [9], 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 in seconds on a customer GPU.
- It **covers multiple use cases**. We estimate gradients for meshes, displacement mapping, Catmull-Clark subdivision surfaces [10], semi-transparent geometry, physically based materials, 3D volumetric data and 3D Gaussian Splats [11].

[1]Intel Labs, Grenoble, FRANCE. Correspondence to: Thomas Deliot <thomas.deliot@intel.com>.

Published at the 2^{nd} Differentiable Almost Everything Workshop at the 41^{st} International Conference on Machine Learning, Vienna, Austria. July 2024. Copyright 2024 by the author(s).

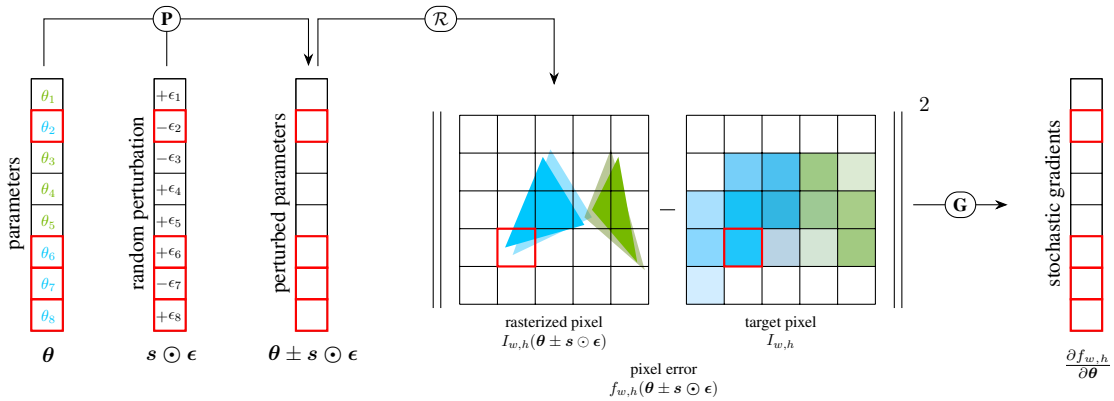

Figure 1: **Overview of our differentiable rasterizer.** The first compute shader (**P**) perturbs the scene parameters before they are rasterized ($\mathcal{R}$). The second compute shader (**G**) accumulates the error differences, which provide a gradient estimate. The key point of our approach is that it accumulates the contribution of a pixel (in red in the images) only in its contributing parameters (in red in the vectors).

2. Per-Pixel Stochastic Gradient Estimation

Our per-pixel method builds on *Stochastic Gradient Estimation* to differential rasterization. There, the objective is to optimize a 3D scene such that a rasterized 2D image from this scene matches a target image. To do this, we need to estimate the gradients of the rasterization computations.

2.1. Stochastic Gradient Estimation

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

Our goal is to compute the gradient of the loss function w.r.t. the input parameters: $\frac{\partial f}{\partial \theta_i}$. Instead of relying on automatic differentiation, we estimate the gradient stochastically

$$\widehat{\frac{\partial f}{\partial \theta_i}} = \frac{f(\boldsymbol{\theta} + \boldsymbol{s} \odot \boldsymbol{\epsilon}) - f(\boldsymbol{\theta} - \boldsymbol{s} \odot \boldsymbol{\epsilon})}{2\, s_i\, \epsilon_i}. \quad (1)$$

where $\boldsymbol{\epsilon} = (\epsilon_1, .., \epsilon_d)$ is a user-defined perturbation magnitude vector $\boldsymbol{s} = (s_1, .., s_d)$ is a random sign vector, $s_i \in \{-1, +1\}$, where each sign has equal probability. We note $\boldsymbol{s} \odot \boldsymbol{\epsilon}$ the element-wise product of both vectors.

2.2. Per-Pixel Formulation

The stochastic gradient estimate of Equation (1) is noisy, especially in a high-dimensional parameter space (Figure 2, first row). In our use case, the error of a pixel contributes to every parameter, even if this parameter is never used to compute the pixel's value. Hence, every parameter receives noisy gradients from every pixel, increasing the variance of the estimator. We propose to compute gradients per-pixel to alleviates this problem and makes the method scalable.

Derivation. We assume that the loss function we use (the l_2 error in our applications), is a sum of per-pixel errors:

$$f(\boldsymbol{\theta}) = \sum_{(w,h) \in W \times H} f_{w,h}(\boldsymbol{\theta}), \quad (2)$$

and the gradient can be defined in the same way:

$$\frac{\partial f}{\partial \theta_i} = \sum_{(w,h) \in W \times H} \frac{\partial f_{w,h}}{\partial \theta_i}. \quad (3)$$

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:

$$\frac{\partial f}{\partial \theta_i} = \sum_{(w,h) \text{ impacted by } \theta_i} \frac{\partial f_{w,h}}{\partial \theta_i}. \quad (4)$$

By applying the estimator of Equation (1) to Equation (4) we obtain the stochastic gradient estimate our method is based on:

$$\widehat{\frac{\partial f}{\partial \theta_i}} = \sum_{(w,h) \text{ impacted by } \theta_i} \widehat{\frac{f_{w,h}}{\partial \theta_i}}. \quad (5)$$

In Section 3, we show how to implement this equation with a rasterizer and compute shaders.

3. Turning a Rasterizer Differentiable

We turn a rasterizer based renderer (denoted $\mathcal{R}$) to a differentiable one with 2 compute shaders: a **P** (perturbation) and a **G** (gradient) shaders which implements Equation (5) We provide an overview of our pipeline in Figure 1 and detail its three steps next.

3.1. Perturbation (Compute shader P)

First, we launch a compute shader that execute Algorithm 1 over d threads (one thread per scene parameter). The shader computes the perturbed scene parameters $\boldsymbol{\theta} + \boldsymbol{s} \odot \boldsymbol{\epsilon}$ and $\boldsymbol{\theta} - \boldsymbol{s} \odot \boldsymbol{\epsilon}$ and store them in GPU memory. Its main ingredient is the generation of the random sign vector $\boldsymbol{s}$ via **randomsign**(), which we implement with a random hash function [12].

Algorithm 1 Compute shader **P** (perturbation)

Require: thread ID i
  load θ_i, ϵ_i ▷ load 2 float
  s_i = **randomsign**() ▷ hash function [12]
  store $s_i\epsilon_i$, $\theta_i + s_i\epsilon_i$, $\theta_i - s_i\epsilon_i$ ▷ store 3 float

3.2. Rasterization ($\mathcal{R}$)

Then we utilize the perturbed parameter to rasterize the scene twice. First, using parameters $\boldsymbol{\theta} + \boldsymbol{s} \odot \boldsymbol{\epsilon}$ and then using parameters $\boldsymbol{\theta} - \boldsymbol{s} \odot \boldsymbol{\epsilon}$. We thus obtain two images: $I(\boldsymbol{\theta} + \boldsymbol{s} \odot \boldsymbol{\epsilon})$, and $I(\boldsymbol{\theta} - \boldsymbol{s} \odot \boldsymbol{\epsilon})$. Those are used to evaluate the gradient in the next shader.

3.3. Gradients Estimate (Compute shader G)

Finaly, we launch a compute shader that execute Algorithm 2 over $W \times H$ threads (one thread per pixel). The shader computes the pixel errors $f_{w,h}(\boldsymbol{\theta} + \boldsymbol{s} \odot \boldsymbol{\epsilon})$ and $f_{w,h}(\boldsymbol{\theta} - \boldsymbol{s} \odot \boldsymbol{\epsilon})$ between the perturbed-scene images $I(\boldsymbol{\theta} + \boldsymbol{s} \odot \boldsymbol{\epsilon})$ and $I(\boldsymbol{\theta} - \boldsymbol{s} \odot \boldsymbol{\epsilon})$ 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 (5). 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): $\mathcal{R}$ has to output an ID buffer (else, we get it from another rasterization step).

Algorithm 2 Compute shader **G** (gradient)

Require: thread ID (w, h)
  load $I_{w,h}(\boldsymbol{\theta} + \boldsymbol{s} \odot \boldsymbol{\epsilon})$, $I_{w,h}(\boldsymbol{\theta} - \boldsymbol{s} \odot \boldsymbol{\epsilon})$, $I_{w,h}$ ▷ load 3 float3 (3× rgb)
  $f_{w,h}(\boldsymbol{\theta} + \boldsymbol{s} \odot \boldsymbol{\epsilon}) = \|I_{w,h} - I_{w,h}(\boldsymbol{\theta} + \boldsymbol{s} \odot \boldsymbol{\epsilon})\|^2$
  $f_{w,h}(\boldsymbol{\theta} - \boldsymbol{s} \odot \boldsymbol{\epsilon}) = \|I_{w,h} - I_{w,h}(\boldsymbol{\theta} - \boldsymbol{s} \odot \boldsymbol{\epsilon})\|^2$
  for each parameter θ_i contributing to pixel (w, h) **do** ▷ implementation of Equation (5)
    load $s_i\epsilon_i$ ▷ load 1 float
    AtomicAdd$\left(\frac{\partial f}{\partial \theta_i} \leftarrow \frac{f_{w,h}(\boldsymbol{\theta}+\boldsymbol{s}\odot\boldsymbol{\epsilon}) - f_{w,h}(\boldsymbol{\theta}-\boldsymbol{s}\odot\boldsymbol{\epsilon})}{2\, s_i\, \epsilon_i}\right)$ ▷ atomic add 1 float
  end for

4. Results

We provide results of our method and compare it to a vanilla stochastic difference approach. For all our example, we modified Unity to apply our compute shaders and the Adam optimizer (as a compute shader) once the gradients are computed. All our results are timed on an NVIDIA 4090 GPU.

Optimizing Triangle Meshes. In Figure 2 we illustrate the difference between the *full-image* approach of Equation (1) and the *per-pixel* approach of Equation (5). In this experiment, each triangle is represented by 12 parameters (3 vertices + 1 RGB color). Using 100K triangles, we obtain a total of 1228800 parameters. Optimizing with the *full-image* error is impractical with that numbers of parameters. The *per-pixel* gradients permit a quick convergence.

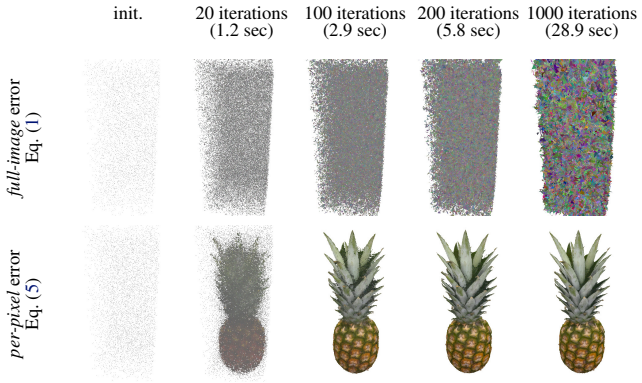

Figure 2: **Validation of the per-pixel formulation.** In this experiment, we optimize triangles soups to match a 2D image. The *full-image*variant implements Equation (1) where the error over the whole image contributes to every parameter and the *per-pixel* approach implements Equation (5).

Optimizing various primitives. Figures 3 showcase optimizing subdivision surfaces [10], PBR textures, volumes, and Gaussian Splats [11]. For Catmull-Clark subdivision surfaces [10], we optimize the control mesh and tessellate it on the fly in the rasterizer [13]. We write the control mesh's triangle in the ID buffer. We additionaly optimize the displacement and normal maps and further store UV coordinates. For physically based shading, we optimize roughness, metallicity, albedo, height and normal maps. Similary to the subdivision surface case, we need to store the UV coordinate in the ID buffer. For 3D Gaussian splats, we rasterize transparent quads and use a front-to-back sorting to output a deep ID buffer. To improve the results, we implement an additional resampling and a splat subdivision compute shader executed after each gradient descent, following Kerb et al. [11]. Lastly, for 3D volumes, we rely on an additional ray marching phase in compute shader **G** to splat the gradients and avoid to store all the voxels coordinates.

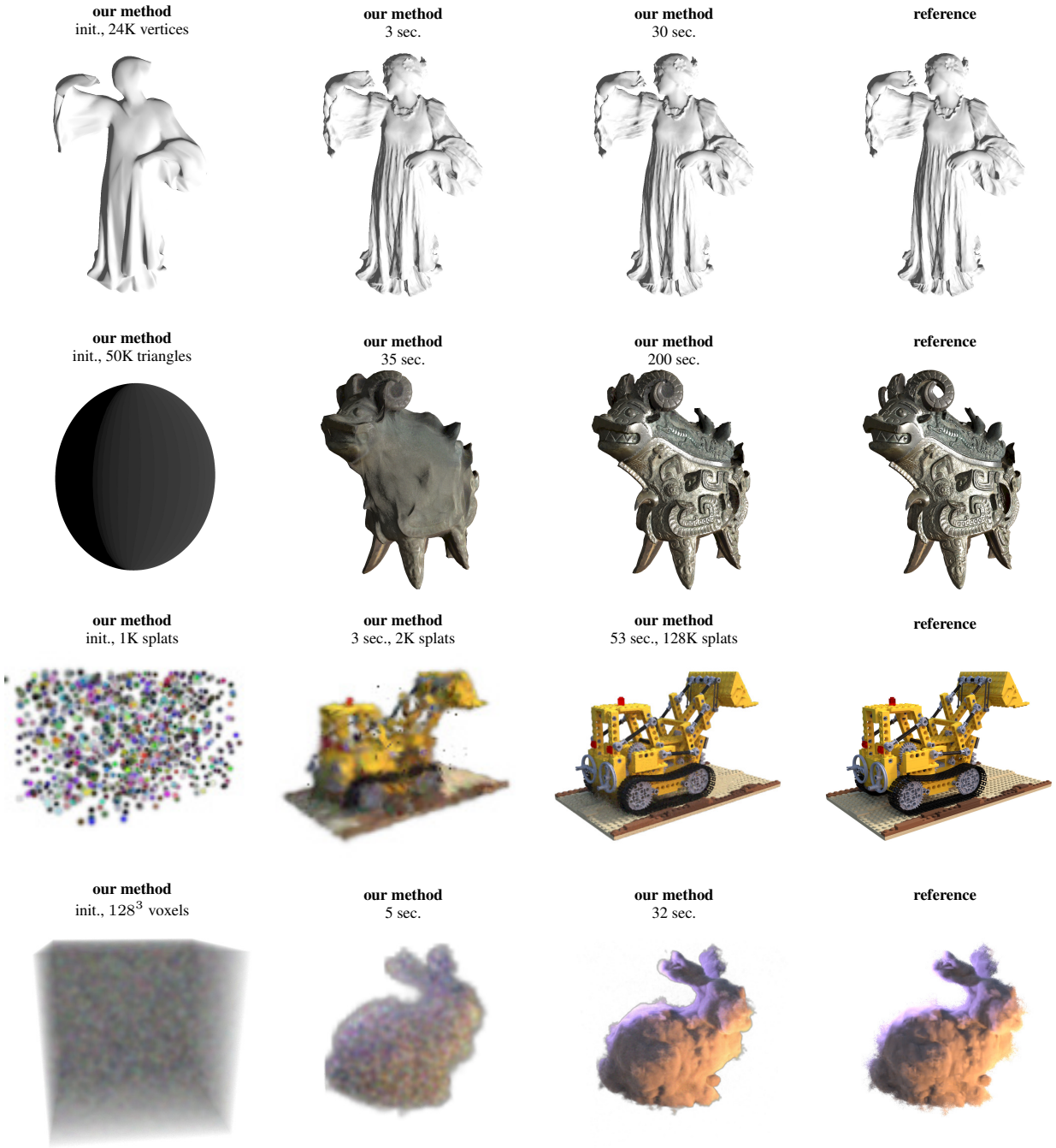

Figure 3: **Optimizing 3D Assets.** We showcase different application of our method. We can optimize either subdivision surfaces, PBR materials, 3D Gaussian Splats, and voxels.

5. 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 meshes, subdivision surfaces, physically based materials, volumes, and 3DGS.

However, we do not position our method as a replacement for other state-of-the-art differentiable rasterizers. We aim to bring the benefits of differentiable rasterization to an audience that already possesses a 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 within the existing engine.

References

[1] H. Rhodin, N. Robertini, C. Richardt, H.-P. Seidel, and C. Theobalt, "A versatile scene model with differentiable visibility applied to generative pose estimation," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 765–773.

[2] H. Kato and T. Harada, "Learning view priors for single-view 3d reconstruction," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 9778–9787.

[3] S. Wu, C. Rupprecht, and A. Vedaldi, "Unsupervised learning of probably symmetric deformable 3d objects from images in the wild (invited paper)," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 4, pp. 5268–5281, 2023.

[4] J. Hasselgren, J. Munkberg, J. Lehtinen, M. Aittala, and S. Laine, "Appearance-driven automatic 3d model simplification.," in *EGSR (DL)*, 2021, pp. 85–97.

[5] D. Azinovic, T.-M. Li, A. Kaplanyan, and M. Nießner, "Inverse path tracing for joint material and lighting estimation," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 2447–2456.

[6] W. Jakob, S. Speierer, N. Roussel, and D. Vicini, "Dr.jit: A just-in-time compiler for differentiable rendering," *Transactions on Graphics (Proceedings of SIGGRAPH)*, vol. 41, no. 4, 2022.

[7] S. Bangaru, L. Wu, T.-M. Li, *et al.*, "Slang.d: Fast, modular and differentiable shader programming," *ACM Transactions on Graphics (SIGGRAPH Asia)*, vol. 42, no. 6, pp. 1–28, Dec. 2023.

[8] S. Laine, J. Hellsten, T. Karras, Y. Seol, J. Lehtinen, and T. Aila, "Modular primitives for high-performance differentiable rendering," *ACM Transactions on Graphics*, vol. 39, no. 6, 2020.

[9] M. Fu, "Stochastic gradient estimation," *Technical report*, 2005.

[10] E. Catmull and J. Clark, "Recursively generated b-spline surfaces on arbitrary topological meshes," *Computer-Aided Design*, vol. 10, no. 6, pp. 350–355, 1978.

[11] B. Kerbl, G. Kopanas, T. Leimkuehler, and G. Drettakis, "3d gaussian splatting for real-time radiance field rendering," *ACM Trans. Graph.*, vol. 42, no. 4, 2023.

[12] M. Jarzynski and M. Olano, "Hash functions for gpu rendering," *Journal of Computer Graphics Techniques (JCGT)*, vol. 9, no. 3, pp. 20–38, Oct. 2020, ISSN: 2331-7418.

[13] J. Dupuy and K. Vanhoey, "A halfedge refinement rule for parallel catmull-clark subdivision," *Computer Graphics Forum*, vol. 40, no. 8, pp. 57–70, 2021.