DOI: 10.1111/cgf.14455
EUROGRAPHICS 2022 / R. Chaine and M. H. Kim
(Guest Editors)

Volume 41 (2022), *Number 2*

Dressi: A Hardware-Agnostic Differentiable Renderer with Reactive Shader Packing and Soft Rasterization

Yusuke Takimoto*, Hiroyuki Sato*, Hikari Takehara*, Keishiro Uragaki,
Takehiro Tawara, Xiao Liang, Kentaro Oku, Wataru Kishimoto, and Bo Zheng

Huawei Technologies Japan K.K.

Figure 1: *Applications of our differentiable renderer called Dressi. From left to right: optimization of geometry and material (a pear and a cow), normal map, environment map and material, and hair material of the digital human. The applications work on all devices supporting a modern graphics API Vulkan [Khr21] such as mobiles, tablets, cloud servers, laptops, and desktop PCs.*

Abstract

Differentiable rendering (DR) enables various computer graphics and computer vision applications through gradient-based optimization with derivatives of the rendering equation. Most rasterization-based approaches are built on general-purpose automatic differentiation (AD) libraries and DR-specific modules handcrafted using CUDA. Such a system design mixes DR algorithm implementation and algorithm building blocks, resulting in hardware dependency and limited performance. In this paper, we present a practical hardware-agnostic differentiable renderer called Dressi, which is based on a new full AD design. The DR algorithms of Dressi are fully written in our Vulkan-based AD for DR, Dressi-AD, which supports all primitive operations for DR. Dressi-AD and our inverse UV technique inside it bring hardware independence and acceleration by graphics hardware. Stage packing, our runtime optimization technique, can adapt hardware constraints and efficiently execute complex computational graphs of DR with reactive cache considering the render pass hierarchy of Vulkan. HardSoftRas, our novel rendering process, is designed for inverse rendering with a graphics pipeline. Under the limited functionalities of the graphics pipeline, HardSoftRas can propagate the gradients of pixels from the screen space to far-range triangle attributes. Our experiments and applications demonstrate that Dressi establishes hardware independence, high-quality and robust optimization with fast speed, and photorealistic rendering.

CCS Concepts

• ***Computing methodologies*** $\rightarrow$ ***Rasterization; Shape inference;***

1. Introduction

Inverse rendering is a long-standing problem in estimating scene attributes from 2D images in computer graphics and computer vision fields. Differentiable rendering (DR) plays an important role in applications, such as facial geometry reconstruction [EST*20; GPKZ19; GPKZ21] and camera pose estimation [RRR*15]. It recovers scene parameters by the gradient propagation of the loss functions defined between the rendered and observed images. For example, it is possible to estimate materials if the appearance and other scene settings are already known [ALKN19].

DR can be grouped into two categories: ray tracing [LADL18; NVZJ19] and rasterization [KUH18; CGL*19; VKP*19]. The ray

* These authors contributed equally to this work

tracing-based methods consider complex effects generated by light transport simulations, such as indirect illumination and polarization [NVZJ19]. They achieve the best rendering quality and accuracy with heavy computation. In contrast, rasterization-based methods require fewer computations and provide more efficient solutions, attracting much attention for practical scene reconstruction. We focus on the rasterization-based methods in this paper.

Most existing rasterization-based DR systems ignore hardware dependencies. They are designed to run on high-end hardware from specific vendors (e.g., NVIDIA), and they often implement the DR algorithm using automatic differentiation (AD) libraries for neural networks (e.g., PyTorch and TensorFlow) and additional modules written in the general-purpose graphical processing unit (GPGPU) API, which relies on CUDA for most cases. Because some DR-specific and performance-critical functions (e.g., rasterization and texture sampling) involve pixel-wise operations, writing them down in the DR layer using an efficient batch operation with AD libraries is difficult. Such a tightly coupled DR system design increases hardware dependency and limits performance. The AD libraries are not optimized for complex computational graphs inherent to DR. Moreover, DR systems cannot optimize performance at the border between the AD libraries and handwritten CUDA modules. Therefore, it is not easy to run the existing DR systems on graphics hardware from various vendors (e.g., Intel, AMD, and Arm) at practical speeds, especially on low-end edge devices. Communicating with a remote server that has a DR system installed is an option; nonetheless, it impairs the interactivity of real-time applications, owing to latency and raises privacy issues. Making a DR work efficiently on low-end hardware is a challenging problem.

To obtain hardware independence for DR systems, we can use the graphics pipeline API, which has hardware rasterizers and shaders, instead of the GPGPU API. However, propagating gradients in a screen space under the limitations of graphics pipelines is another challenging problem. The existing rasterization-based DR methods generate gradients in the screen spaces using the special rendering processes easily implemented by CUDA, which allows developer descriptions with high degrees of freedom; however, their efficient implementation using a graphics pipeline is difficult. For instance, SoftRasterizer (SoftRas) [LLCL19] computes distances between pixels and the edges of projected triangles in a pixel-wise manner. Nevertheless, pixel-wise computation considering all triangles is not easy for hardware rasterizers. Nvdiffrast [LHK*20] employs analytic anti-aliasing (AA), which considers edge information among geometrically neighboring triangles. Unfortunately, accessing mesh topology information is also difficult for shaders. Therefore, we cannot port the existing methods to the graphics pipeline.

We propose Dressi, a practical rasterization-based differentiable renderer that solves the above problems. Dressi is based on a new design that completely separates the AD layer and DR algorithms. DR algorithms are fully described by the AD layer and receive hardware independence and performance optimization through the overall system. Our AD layer is Dressi-AD, which is implemented in Vulkan [Khr21], a vendor-independent graphics pipeline API. Furthermore, it is tailored to the DR to support all its primitive operations, including rasterization and texture sampling. The inverse UV technique in Dressi-AD carefully handles the backward pass of hardware texture sampling. At runtime, our stage packing dynamically converts the computational graphs into optimal execution orders to the command buffers. It maximizes hardware performance by considering the constraints for the render pass and subpass of the running hardware. Static values on computational graphs are often observed in DR, such as rasterization results with static geometries. Our reactive cache embedded in the stage packing automatically detects the static values and caches them to skip later computations.

Furthermore, we propose a new rasterization-based DR method, HardSoftRas, to realize far-range gradients from the screen space to vertex attributes, such as SoftRas, under the limitations of a graphics pipeline. In this paper, we use the term, *far-range* gradient, if the DR system propagates the gradient at a rendered pixel to vertices projected onto far positions in screen space (i.e., more than one pixel away). Otherwise, we call it *near-range* gradient. The key to the gradient backward in the screen space is the pixel-to-triangle distance computation; however, no prior art has implemented it on a graphics pipeline owing to the difficulty in handling the association between pixels and triangles. Our face-wise approach, implemented on an inflexible graphics pipeline, successfully updates the pixel-wise computation of the existing methods written in flexible CUDA. Furthermore, a new depth-shift method is proposed to fit a few buffers generated by the hardware rasterizer, whereas the existing methods rasterize a large number of buffers using software. We also show that HardSoftRas is a natural extension of anti-aliasing, which is often used in real-time forward rendering.

We experimentally show that Dressi behaves identically on various hardware shipped from different vendors (e.g., NVIDIA, AMD, Intel, and Arm) with varying performance, ranging from servers to mobile platforms. We also validate that our stage packing and reactive cache increase forward and backward speeds. Moreover, we compare our method to state-of-the-art rasterization-based differentiable renderers to validate HardSoftRas and the overall system. Dressi achieves better speed and quality with both synthetic and real data. Finally, we show that our method can render a photorealistic digital human and optimize hair parameters, which are not supported by existing DR systems.

2. Related Work

2.1. Ray Tracing-Based Differentiable Rendering

Ray tracing-based DR can simulate complex light transport models with visibility handling. Redner [LADL18] established a method for complex inverse problems of scene parameters based on Monte Carlo ray tracing [Kaj86]. Mitsuba 2 [NVZJ19] can simulate the complicated light transport phenomena such as scattering, polarization, and spectroscopy. Moreover, it achieves highly efficient computations with template meta-programming for various data types and a retargetable just-in-time (JIT) compilation for AD. Radiative backpropagation [NSRJ20] is a memory-efficient adjoint approach that reduces the computation of backward functions in continuous light transportation. Path-space DR [ZMY*20] shows better efficiency using the new Monte Carlo estimators. Path replay backpropagation [VSJ21] proposes a ray tracing-based megakernel generation that maintains linear time complexity with constant memory usage. Ray tracing-based methods focus on photo-realistic results and accurate backpropagation. Although they have shown remarkable speedups recently, they still require exceedingly high computational costs for low-end devices lacking hardware ray tracers. Some code optimization techniques that reduce heavy computations on high-end hardware [NVZJ19; VSJ21] are related to our stage packing process. However, ours differs from existing techniques because it is designed to adapt to varying graphics hardware constraints.

2.2. Rasterization-Based Differentiable Rendering

Dasterization-based DR [LB14; KUH18; CGL*19; VKP*19] exhibits a faster computational speed than the ray tracing-based DR. SoftRas [LLCL19] proposes shape optimization as a probabilistic process. It computes the minimum pixel-edge distance for all pairs of edges and pixels in the screen space. The distances and depth values are stored in multiple buffers. Then, the distances are converted to probability, and pixel colors are calculated by aggregating the buffers based on the probability and the depth values. Although SoftRas can propagate screen space gradients to far-range vertex attributes, it is difficult to handle large geometries because it requires as many buffers as faces. To improve the scalability against large geometries, PyTorch3D [RRN*20] extends SoftRas by introducing thresholds for the radius of blurring in the screen space and the number of buffers per pixel. During rasterization, PyTorch3D enlarges each face in the screen space within the blur radius from their edges and stores the distances and depth values per pixel in the buffers. The pixel-edge distance computation step of SoftRas and PyTorch3D is implemented as a pixel-wise computation to track the triangle correspondences in CUDA. We cannot port the algorithm to a graphics pipeline because the hardware rasterizer is based on a face-wise calculation. Another option is to use compute shaders; nonetheless, their computational costs are prohibitive, and they are not supported well on mobile platforms.

Nvdiffrast [LHK*20] achieves significant acceleration, identifying modular primitives of DR. It leverages the performance of GPUs, especially hardware rasterization using OpenGL. Nvdiffrast generates screen space gradients via analytic anti-aliasing. In contrast to SoftRas and PyTorch3D, forward rendering of Nvdiffrast retains the original appearance of rendered images. However, its screen space gradients are propagated only within near-range vertices, because anti-aliasing focuses on boundaries. Moreover, its anti-aliasing step handles edge-triangle correspondences in the CUDA code. Operating edge-based data structures is difficult for the shaders of graphics pipelines. Nvdiffmodeling [HML*21] is an optimization method for a mesh and shading model with physically-based shading over Nvdiffrast. Nvdiffmodeling reconstructs transparency using depth peeling [Eve01], but depth peeling is not used for mixing far-range triangle attributes, as in our study. A survey paper [KBM*20] summarizes recent DR methods.

2.3. Automatic Differentiation

AD automatically calculates function derivatives. Advances in the usability of AD [GW08; AM*10] gives rise to new concepts of differential programming, such as DR. AD on GPU utilizes implementation techniques such as shader code generation [Lat08; Fuj08; GRF11; Mur12; DMZ*17; HFF18] and kernel fusion [WLY10; OUN*17]. Checkpointing [Gri92] is a method that reduces memory usage in a reverse-mode AD. Although recent deep learning compilers utilize compilation techniques [LLL*21], they do not sufficiently support general differentiable programming. Most deep networks are dense and stationary because they consist of block-shaped modules. In contrast, computational graphs in other fields, such as DR and inverse physics simulation [HAL*20], can be complicated, sparse, and dynamic. Most rasterization-based DRs develop DR algorithms on AD libraries for deep learning, such as PyTorch [PGM*19] and TensorFlow [MAP*15], and additionally implement manual CUDA modules for DR-specific rasterization and texture sampling. Therefore, their performance is not well optimized. Unique JIT compilation methods are proposed in each domain. Mitsuba 2 [NVZJ19] has proposed the computational graph simplification for DR that repeats kernel fusion before JIT compilation. AsyncTaichi [HXKD20] proposes asynchronous front-end JIT optimization for dynamic optimization tasks. From the viewpoint of supporting platforms, standard AD libraries require GPGPU, particularly CUDA, and they can run on a CPU. This indicates that they do not consider desktop PCs and laptops with other vendors' GPUs and mobiles. Mitsuba 2 achieves stronger hardware independence, thanks to its own JIT compilation. It works either on a GPU with CUDA or various CPUs with single instruction/multiple data (SIMD) operations. However, they do not consider other GPU platforms. We perform a JIT compilation of AD for DR applications that works on all modern GPUs. TensorFlow.js [STA*19] has an OpenGL Shading Language (GLSL) backend for cross-platform browsers, but its performance for DR is limited because it is designed for neural networks. Enzyme [MC20] supports AD of low-level virtual machine (LLVM) codes. Enzyme with a translator between LLVM and standard portable intermediate representation (SPIR)-V would be an option for hardware-agnostic DR, but SPIR-V does not contain a hardware rasterizer. Thus, it is difficult for Enzyme to establish a high-performance DR.

3. Dressi

We depict the pipeline of our proposed Dressi in Fig. 2. Dressi-AD describes all the components of our DR algorithms. Dressi takes inverse problems as inputs and outputs optimized results. It consists of several components, such as Renderer, Optimizer, and Loss function. The core component is Renderer, which applies HardSoftRas after scene updates (e.g., animation). In the following subsections, we describe the goals and designs for establishing this pipeline. Then, we introduce the implementation and validation of our pipeline. The implementation details are provided in the supplemental material.

3.1. System Goals

This paper describes Dressi, a high-performance differentiable renderer for practical usage that supports a wide range of hardware. In short, the proposed Dressi concept is "fully written in graphics pipeline-based AD for DR". As shown in Table 1, the existing rasterization-based DR frameworks use general-purpose AD

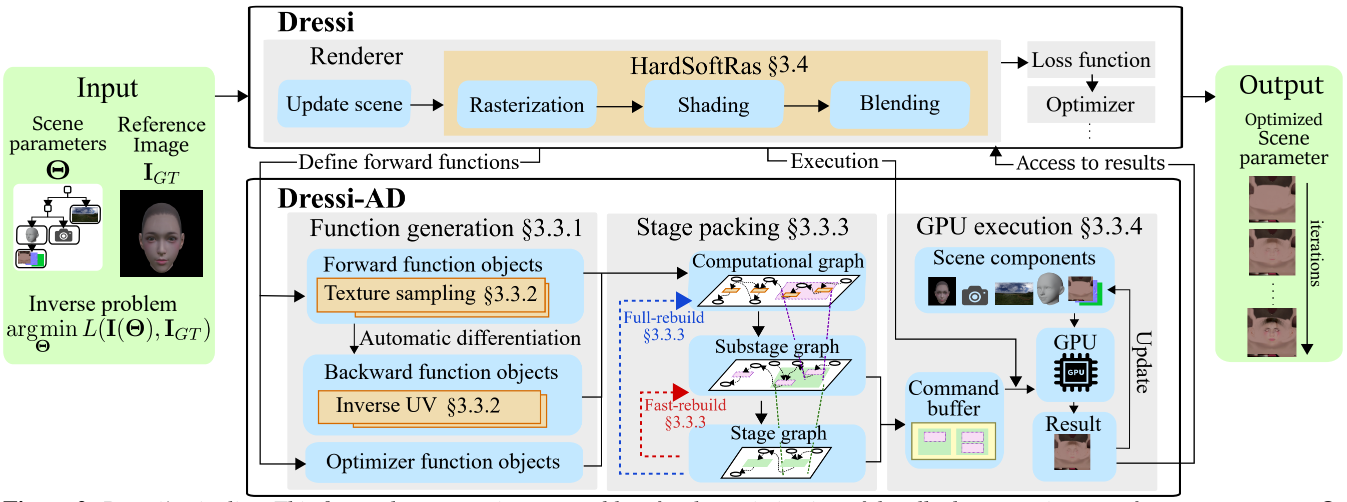


Figure 2: *Dressi's pipeline. This figure shows a toy inverse problem for the optimization of the albedo texture in a set of scene parameters, Θ, to fit the rendered result to the reference image, $\mathbf{I}_{GT}$. Dressi is written with forward functions of Dressi-AD. Dressi-AD builds a computational graph and executes inverse rendering on the GPU via function generation and stage packing.*

	SoftRas [LLCL19]	PyTorch3D [RRN*20]	Nvdiffrast [LHK*20]	**Ours**
Graphics Hardware	NVIDIA	NVIDIA	NVIDIA	NVIDIA, AMD, Intel, Arm, ...
AD	PyTorch	PyTorch	PyTorch and TensorFlow	Dressi-AD
	Implementation / Backend API			
Math	AD / CUDA	AD / CUDA	AD / CUDA	AD / Vulkan
Rasterization	Manual / CUDA	Manual / CUDA	Manual / CUDA + OpenGL	AD / Vulkan
Texture sampling	Manual / CUDA	AD / CUDA	Manual / CUDA	AD / Vulkan

Table 1: *Comparison of rasterization-based DR frameworks. Thanks to our full AD approach with Dressi-AD based on Vulkan, ours can work on various graphics hardware and have high performance.*

and supplement DR-specific functions using handwritten modules in the GPGPU API. This typical design is not scalable to various graphics hardware and can cause performance degradations owing to a lack of optimization across the entire system. To solve these problems, we set the design goals of Dressi as follows:

- G1: **Hardware-agnostic**. To democratize DR and explore the possibility of its practical application in everyday life, the DR system must be hardware-independent. People use various hardware, including low-end mobile devices, but existing DR systems do not support most of them.
- G2: **Hardware-accelerated**. Acceleration by graphics hardware is practically essential for handling massive DR computations.
- G3: **Adaptivity**. Automatic and dynamic adaptation to running hardware to boost performance is desirable for a practical DR system. Naive implementation with a static pipeline can cause deterioration of efficiency for some hardware.
- G4: **Ease of modification**. A practical DR system should be easily customized with APIs that support all primitive operations for DR. Unlike traditional renderers, many optimization settings, not just shaders, must be set for each DR application. DR algorithms should be easily customized as well to incorporate the latest DR techniques.
- G5: **Independent**. Practical DR systems should support independent edge devices without communicating with remote servers. For real-time applications with user interaction, the latency caused by communications should be avoided. Moreover, DR plays a vital role in appearance modeling using personal information (e.g., human face). Such modeling algorithms should be performed on edge devices with local data to protect user privacy.
- G6: **Optimization friendly**. Optimization with images is a primitive purpose of DR. DR should be capable of robust and fast optimization.

3.2. System Design

To achieve our goals, we make the following design choices:

- C1: **Graphics pipeline only** (⇒ G1 and G2). We build all components of our system on a standard graphics pipeline. Consequently, our Dressi obtains cross-platform properties and acceleration by graphics hardware.
- C2: **Monolithic system** (⇒ G4 and G5). We design our AD and Dressi as a monolithic system. Considering only our AD, users can quickly develop DR applications and update Dressi itself. An independent device completes the DR execution.
- C3: **Runtime optimization** (⇒ G3). Hardware capability varies across devices (e.g., the number of input attachments of Vulkan). Our system dynamically generates and optimizes shader codes at runtime and maximizes the performance of each device.
- C4: **Fully written in AD** (⇒ G2, G3 and G4). Our DR algorithms are fully written in our AD for DR. Thus, the DR al-

gorithms are wholly separated from the AD. The APIs of our AD provide all primitive operations for DR, such as rasterization and texture sampling. Our AD globally optimizes Dressi's performance. Users can extend Dressi simply to write forward passes with the functions of the AD.

- C5: **Far-range gradient** (⇒ G6). Our rendering process is designed to deliver far-range gradients from the screen space to vertex attributes. It enhances fast and robust convergence.

Based on the design choices above, Dressi is materialized with the following implementations:

- I1: **Dressi-AD** in Section 3.3 (⇒ C1, C2, C3 and C4). A new AD library for DR is proposed. The backend of Dressi-AD is a cross-platform graphics pipeline API, Vulkan. Dressi-AD describes all components of Dressi; therefore, Dressi obtains hardware independence and acceleration.
- I2: **Inverse UV** in Section 3.3.2 (⇒ C1 and C4). Regarding the backward computation of texture sampling with hardware in our AD, we propose an inverse UV texture.
- I3: **Stage packing** in Section 3.3.3 (⇒ C2 and C3). We propose a JIT compilation technique for Vulkan's hierarchy to map multiple subpasses to a render pass. Shader codes on the computational graph are automatically packed into optimal GPU execution units, considering hardware constraints. Reactive cache is integrated to skip redundant computation.
- I4: **HardSoftRas** in Section 3.4 (⇒ C1 and C5). Contrary to most DR systems, ours is fully implemented under the limitation of a graphics pipeline. HardSoftRas is a novel rendering process enabling far-range gradient with graphics hardware.

3.3. Dressi-AD: A Vulkan-Based AD Library for DR

Dressi-AD, a hardware-agnostic AD for DR built on Vulkan, is the foundation of our system. OpenGL was another candidate for the graphics pipeline, but it is too abstract to exploit the performance of modern hardware. OpenCL and compute shaders may not be supported well on mobile devices, and they cannot use hardware rasterizers. Moreover, their performances are limited since there is no correspondence to Vulkan's subpasses. We prefer Vulkan because of its low-level APIs and the potential to increase the rendering speed. Unlike multi-dimensional tensors used in most existing AD libraries, we choose 2D images as the primitives of our AD, considering compatibility with graphics pipeline and DR. Dressi-AD has all primitive operations for DR, including rasterization and texture sampling. Dressi-AD is written in C++17.

Dressi-AD adopts its own variable and function objects for reverse-mode AD. The APIs provided in general languages are the preferred design for developers [MGAK03]. Developers write mathematical problems with the APIs, then Dressi-AD builds a computational graph and executes it on GPUs using the define-and-run scheme. An example of a user program is shown in the supplemental material. The bottom part of Fig. 2 is the build and execution pipeline of Dressi-AD after users define the problems. In this subsection, we explain three key operations in the pipeline: function generation, stage packing, and GPU execution. We use Vulkan terms to introduce a specific implementation.[*]

3.3.1. Function Generation

Dressi-AD instantiates function objects and variable objects in programs written by users. Each forward function object has the GLSL representation that is compatible with a fragment shader, and it has a method to generate backward function objects. This GLSL code generation step roughly follows TensorFlow.js [STA*19]. Some function objects for DR cannot be defined using the built-in GLSL functions. One such exception is the backward functions of texture sampling, which will be explained in Section 3.3.2.

The variable object is a data structure for the inputs and outputs of the functions. For reasons described later, we need to split the shaders and output the intermediate variables from them. Vertex shaders cannot export intermediate variables, and compute shaders are not well optimized for graphics purposes. Therefore, we prefer fragment shaders with dummy vertex shaders as much as possible, which are commonly used in deferred rendering. In our implementation, we convert the variables into `VkImage`. They can contain any 2D array resources, such as textures, vertex buffers, and blend weight matrices. Note that the variables can represent higher dimensional tensors as stacked 2D arrays.

3.3.2. Inverse UV: Backward for Hardware Texture Sampler

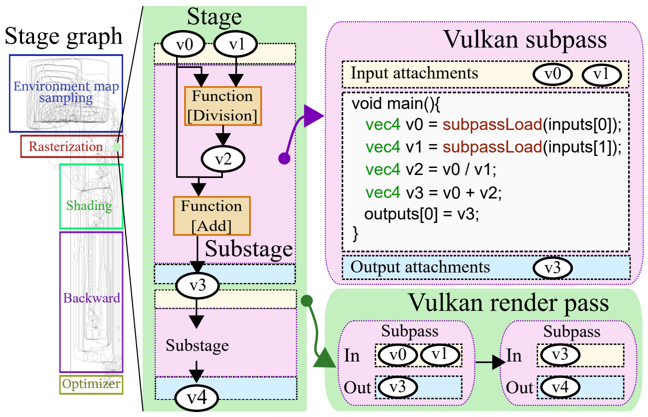
Existing rasterization-based DR systems implement texture sampling with software, for instance, by tensor operation in general-purpose AD libraries [RRN*20] and handwritten CUDA modules [LHK*20]. In contrast, our texture sampling is a function of our AD based on a graphics pipeline. Thus, we are interested in directly handling textures of graphics pipeline. We use `texture()`, which is a function for texture sampling in GLSL with hardware acceleration, as a forward function object to map texture space to screen space. Forward texture sampling with linear interpolation generates a pixel value from four neighboring pixels. In the backward pass, the gradients from the four neighbors should be summed up. However, a naive implementation is difficult because the current APIs of Vulkan do not fully support atomic float add operations.

Therefore, we propose an inverse UV texture, the lookup table for converting gradients from the screen space into texture space. Algorithm 1 calculates it in a compute shader in Vulkan. The inverse UV texture can only maintain the last updated values for each texel, and a constant sampling order may lead to an imbalance. Some texels kept in the inverse UV are updated in every frame, whereas others are never updated. Consequently, artifacts may appear in an optimized texture. To solve this imbalance, we disrupt the sampling order by adding quasi-random numbers generated from a Sobol sequence [Sob67] to p_i^{ss} for each frame. The gradient can be propagated into four texels during iterative optimization. An inverse UV map that samples 3D scenes efficiently for ray tracing-based DR [NDJK21] was recently proposed. In contrast, ours is designed to handle the backward pass with a hardware texture sampler.

[*] `VkImage` is a structure that handles 2D arrays in Vulkan. `VkRenderPass` is a structure that contains subpasses, I/O structures, and dependencies between subpasses. The details are provided in https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/

Algorithm 1 Calculating an inverse UV texture

```
Input: the size $(W_{ss}, H_{ss})$ of screen space, a rasterized UV image
    $\mathbf{I}_{uv} \in \mathbf{R}^{W_{ss} \times H_{ss} \times 2}$ in screen space, the size $(W_{tex}, H_{tex})$ of texture space
Output: an inverse UV texture, $\mathbf{I}_{inv\_uv} \in \mathbf{R}^{W_{tex} \times H_{tex} \times 2}$
1: for all pixel indices, $i = 1, 2, ..., W_{ss} \times H_{ss}$ do
2:     calculate a pixel position in screen, $p_i^{ss}$
3:     fetch and normalize UV value, $p^{tex}$ at $p_i^{ss}$ from $\mathbf{I}_{uv}$
4:     Store $p_i^{ss}$ as a pixel value at $p^{tex}$ in $\mathbf{I}_{inv\_uv}$
5: end for
```



```
void main(){
  vec4 v0 = subpassLoad(inputs[0]);
  vec4 v1 = subpassLoad(inputs[1]);
  vec4 v2 = v0 / v1;
  vec4 v3 = v0 + v2;
  outputs[0] = v3;
}
```

Figure 3: *Stage packing visualization. An example of a stage graph is shown on the left. A stage of rasterization is magnified from the middle to the right. Function objects are packed into a substage, and substages are packed into a stage. v0, ..., v4 are variable objects. A substage corresponds to a subpass, and a stage is converted into a render pass in Vulkan. One subpass has one fragment shader, whose code is generated from the functions objects in a substage. Variable objects, which are the input and output of a substage, are treated as attachments to a subpass. A render pass depends on input and output attachments among subpasses as Vulkan specification.*

3.3.3. Stage Packing

All variable objects and function objects for the forward pass, the backward pass, and optimizers to connect them constitute a single directed graph structure called a *computational graph*. The graph is not executable in a single fragment shader for the following reasons: (1) Inverse UV computation by a compute shader and rasterization by a hardware rasterizer cannot be executed on fragment shaders. (2) A single shader cannot have different sizes for the output images. (3) There are limitations in the number of input and output variables in one shader. These are typical limitations of graphics pipelines. Therefore, we employ hierarchical shader packing to efficiently execute the graph in a reactive manner.

First, function objects in a computational graph are packed into *substages*, which represent fragment shaders, rasterizations, and compute shaders. Each substage except a compute shader is compatible with a subpass of Vulkan. Substages construct an oriented graph, *substage graph*. At the construction process, substage packing, a runtime optimization for each GPU model to minimize the number of substages with a greedy strategy, is performed. Substage packing searches for a good combination of function objects to resolve hardware constraints and reduce bandwidth consumption. As a result, it brings fewer read/write operations and improved memory efficiency. Our method is inspired by the shader partitioning used in multipass rendering [CNS*02], which was proposed only for the forward pass.

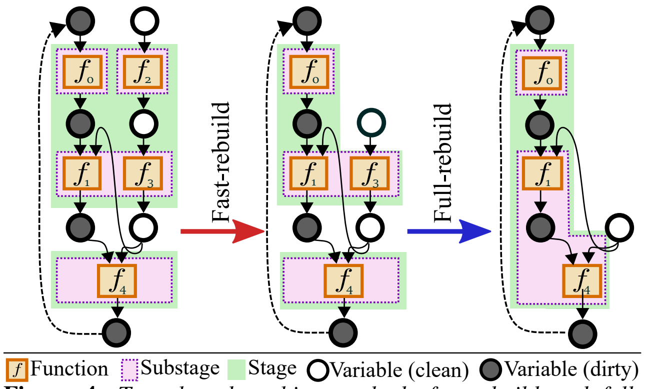


Figure 4: *Two-phased caching methods fast-rebuild and full-rebuild to use clean variable objects as cache. Fast-rebuild packs substages with dirty (i.e., not clean) variable objects as I/O into stages. Full-rebuild packs function objects with dirty variable objects as I/O into substages and constructs stages again. Full-rebuild is slower because it involves updating the shader code, but the graphs reconfigured with full-rebuild run faster than with fast-rebuild.*

After the construction of the substage graph, substages are merged into *stage*s. A *stage graph* is built from the stages for execution as a render pass of Vulkan (i.e., `VkRenderPass`). The two-level hierarchy between a render pass and multiple subpasses allows drivers to improve memory efficiency. Although our graphs may contain compute shaders, Vulkan executes the compute shaders outside the hierarchy. Vulkan has some constraints for render passes. For example, subpasses in the same render pass must have the same image size. Therefore, under these constraints, we apply stage packing to search for a better combination of substages. Stage packing is done similarly as substage packing. Fig. 3 shows the relationship among the function object, substage, and stage.

Our AD instantiates `VkImage` objects only from variable objects exposed beyond the substages, including all scene parameters Θ. Subpasses use the instances as input and output attachments of Vulkan. Function objects in a substage are expanded into GLSL code by adding loading and saving `VkImage`. As shader code simplification [MSPK06], we implement standard optimization techniques such as duplicated function removal and automatic reuse of `VkImage` to reduce memory consumption. Additionally, a two-phased reactive cache is implemented to efficiently handle DR's complex graph structure and partially unchanged values on it. Inspired by reactive programming [BCC*13] and sub-graph culling in a modern game engine [ODo17], our reactive cache automatically detects clean graph nodes that are not updated through optimization and skips their recalculation with cached values that have been computed once. It is especially effective for large blocks such as irradiance sampling and rasterization for a static scene, although they can also work for small snippets. Fig. 4 presents details of our reactive cache. Although we have described the Vulkan implementation, our stage packing can be implemented for any graphics API with a hierarchical structure by minor modifications. Our stage packing is a unique JIT compilation technique for DR because it (1) is designed for AD including backward, (2) handles the two-level hierarchy, and (3) has an embedded reactive cache.

3.3.4. GPU Execution

After stage packing, Vulkan objects are created according to a stage graph and substage graphs. The order of stage execution is recorded into GPU command buffers. Thanks to multiple subpasses in a render pass, Vulkan may automatically reduce bandwidth consumption, depending on the GPU vendor's implementation.

3.4. HardSoftRas: A Hardware Accelerated Soft Rasterizer

We propose a novel rendering process called HardSoftRas to propagate gradients from the screen space to far-range vertex attributes using hardware rasterizers. Moreover, thanks to our full AD approach, the backward step is more efficient than prior art. Dressi-AD can pack shader codes throughout HardSoftRas, although prior art cannot be optimized through common AD libraries and hand-written modules (see Table 1). HardSoftRas consists of three main steps: rasterization, shading, and blending.

3.4.1. Differentiable Rendering

DR is a method that optimizes parameters of a 3D scene by minimizing user-defined metrics between rendered images and ground truth (GT) images. We formulate the scene parameters, Θ, with a set of 3D model, camera, θ_C, lighting, θ_L, and environment, θ_E, parameters. The 3D model parameters are derived from the geometric, θ_G, and material, θ_M, parameters. As Nvdiffrast indicates, rendering is regarded as a function composed of rasterization, shading, and post-processing, which can convert a 3D scene into 2D images. We define the rendering function as follows:

$$\mathbf{I}(\Theta) = f_{render}(\Theta) \tag{1}$$
$$= f_{pp}(f_{shade}(f_{rasterize}(\theta_G, \theta_C), \theta_M, \theta_L, \theta_E)). \tag{2}$$

where f_{render}, $f_{rasterize}$, f_{shade}, and f_{pp} are rendering, rasterization, shading, and post-processing (e.g., anti-aliasing, blending, gamma correction, and masking), respectively. $\mathbf{I}(\Theta) \in \mathbf{R}^{W \times H \times C}$ denotes the rendered image, where $\mathbf{W}$, $\mathbf{H}$, and $\mathbf{C}$ are the width, height, and channel of the image. To make the function differentiable with all parameters of the scene components, the gradients of all the functions, $f_{rasterize}$, f_{shade}, and f_{pp}, should be calculated using the chain rule. For example, given a loss function, $L : \mathbf{R}^{W \times H \times C} \times \mathbf{R}^{W \times H \times C} \to \mathbf{R}$, to calculate the difference between the rendered, $\mathbf{I}(\Theta)$, and reference, $\mathbf{I}_{GT} \in \mathbf{R}^{W \times H \times C}$ images. An optimization problem for Θ can be defined as $\arg\min_{\Theta} L(\mathbf{I}(\Theta), \mathbf{I}_{GT})$. Then, the gradient of Θ can be calculated as $\frac{\partial L}{\partial \Theta} = \frac{\partial L}{\partial \mathbf{I}} \frac{\partial \mathbf{I}}{\partial \Theta}$. The optimization problem can be iteratively solved using a gradient-based method.

Algorithm 2 Rasterization of HardSoftRas

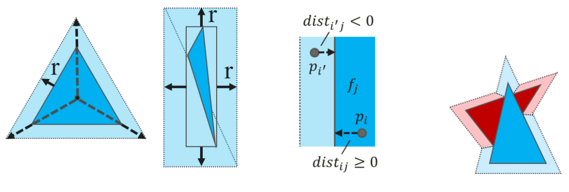
```
Input: The blur radius $\mathbf{r} \in [0,1]$, #buffers $\mathbf{K} \geq 1$, #faces $\mathbf{F} \geq 1$,
       face indices $j \in \mathbf{F}$, and faces $\mathsf{f}_j$
Output: signed distance $dist_i^k$, depth value $depth_i^k$, and the other
       G Buffers $gb_i^k$ at each buffer, $k \in [1, \mathbf{K}]$ and pixel position $p_i$
 1: for $k = 1, 2, ..., \mathbf{K}$ do
 2:     for all face indices $j = 1, 2, ..., \mathbf{F}$ do
 3:         $\mathsf{f}'_j \leftarrow \mathsf{Enlarge}(\mathsf{f}_j, \mathbf{r})$
 4:         Hardware rasterization to generate pixels on $\mathsf{f}'_j$
 5:         for pixel indices $i \in \mathsf{f}'_j$ do
 6:             $dist_{ij} \leftarrow \mathsf{SignedDist}(p_i, \mathsf{f}_j)$
 7:             Compute $depth'_{ij}$ and $gb_{ij}$
 8:             $depth_{ij} \leftarrow \mathsf{Shift}(depth'_{ij}, dist_{ij})$
 9:             Handle occlusion by depth peeling and depth test
10:             Update $dist_i^k$, $depth_i^k$, and $gb_i^k$
11:         end for
12:     end for
13: end for
```

3.4.2. Rasterization

The pixel-edge distance is essential for propagating far-range gradients. SoftRas families handle correspondences between pixels and triangles in pixel-wise CUDA code. Such complex pixel-wise processes cannot be implemented on graphics pipelines. Based on the finding that the pixel-edge distance can be oppositely obtained from enlarged triangles, HardSoftRas utilizes face-wise calculations in the graphics pipeline.

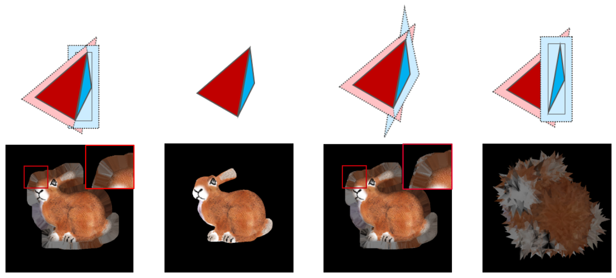
The rasterization step of HardSoftRas is described in Algorithm 2. The outer most loop is for depth peeling with $\mathbf{K}$ buffers. The second outer loop with j rasterizes faces f_j with geometry shaders, and the inner most loop updates the values at a pixel position, p_i, with index i by fragment shaders. Our key contributions are $\mathsf{Enlarge}()$, which enlarges the triangles in the screen space, and $\mathsf{Shift}()$, which updates the depth values of the enlarged face region (*soft face*), considering the original face region (*hard face*).

$\mathsf{Enlarge}()$ makes the projected f_j larger to reach $\mathbf{r}$ from their edges in the geometry shaders. $\mathbf{r} \in [0,1]$ defines the range of blurring in the screen space $\in [0,1]^2$. Fig. 5 (a) depicts this process. For acute projected triangles, we stretch their vertices in the opposite direction of their centroids (see Fig. 5 (a) left). However, this approach does not work well for obtuse triangles because the resulting triangle may become too sharp to cause shaggy boundaries. Thus, we expand the bounding boxes to cover $\mathbf{r}$ if the triangles are obtuse. The expanded bounding boxes are split into pairs of triangles for hardware rasterization (see Fig. 5 (a) right). This paper regards a triangle as obtuse if the smallest angle is less than 30°. Our strategy is less computationally expensive and more robust than the conservative rasterization [HAO05] by keeping the number of triangle primitives as small as possible and avoiding precision errors of floating-point computation in the obtuse triangles. Fig. 6 (a), (b), and (c) show how our $\mathsf{Enlarge}()$ works. For p_i on the expanded face, f'_j, we compute the signed distances, $dist_{ij}$, from the original



(a) Enlarge() **(b)** SignedDist() **(c)** Shift()

Figure 5: *Subroutines used in the rasterization step of HardSoftRas. Triangles in dark blue and red are original hard faces, and triangles in light blue and red are soft faces expanded from the hard faces, respectively. (a) Left:* Enlarge() *for acute triangles. Right:* Enlarge() *for obtuse triangle. (b)* SignedDist() *to compute the signed distance between the pixel position and the edge of a face. (c) Occlusion handling by* Shift()*. The dark blue hard face is in front of the dark red hard face. Soft faces are pushed behind the hard faces, and those closer to the hard faces have closer depth values.*



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

Figure 6: *Enlarge() and Shift() effects. The top row shows the occlusion handling of two adjacent triangles, the blue triangle has a closer depth than red one, and the bottom row shows the blended images with* $\mathbf{K} = 2$*. (a) w/ Enlarge() and w/ Shift(). Smoother soft faces than acute only, (c). This setting is the default. (b) w/o Enlarge(). Normally shaded image. (c) w/ acute only Enlarge() and w/ Shift(). In the magnified red rectangle, boundaries of soft faces are shaggy. (d) w/ Enlarge() but w/o Shift(). Suffering from pseudo-occlusions. In this case, we visualize all faces without an edge mask.*

faces, f_j, by SignedDist(), as shown in Fig. 5 (b). These distances are used to deliver gradients in the screen space during the blending step.

If we expand a soft face by interpolated depth values from its hard face, the soft face may cause artifacts that occlude the necessary hard faces around it. We refer to this problem as *pseudo occlusion* in this paper (see Fig. 6 (d), for instance). In fact, SoftRas and PyTorch3D also have pseudo occlusions; however, this is not a significant problem for them because their blending with numerous buffers (hundreds or thousands) cancels out the artifacts. We want to keep $\mathbf{K}$ small at a maximum of five, due to the linear order of depth peeling. To avoid pseudo occlusions with few buffers, we should assign more importance to hard faces than soft faces. If $dist_i^k \geq 0$, p_i at the kth buffer comes from the hard faces that are inside the original faces. Otherwise, it is located on soft faces. Hence, our depth modification, Shift(), is defined as follows:

$$\mathsf{Shift}(depth, dist) = \begin{cases} 0.5 * depth & \text{if } dist \geq 0, \\ 0.5 * |dist| + 0.5 & \text{if } dist < 0. \end{cases} \quad (3)$$

With $depth \in [0,1]$ and $dist \geq -1$, Shift() places all hard faces in front of all the soft faces. Moreover, soft faces close to the hard faces with a high probability should be rasterized in the buffers. To prioritize such soft faces, we update the depth values of soft faces based on $dist$. Fig. 5 (c) and the top row of Fig. 6 show how Shift() works. We describe the forward pass above. We do not need special care for the backward pass, thanks to our full AD approach.

3.4.3. Shading and Blending

After rasterization, deferred shading is applied to each G buffer gb_i^k to generate a shaded buffer. Dressi supports a variety of shadings such as physically-based shading (PBS), image-based lighting (IBL), skin, and shadows.

Finally, we blend the shaded buffers per pixel to determine the final pixel colors of a shaded image. Our blending is based on SoftRas but we update it in several aspects. First, we calculate the probability $\mathcal{D}_k = \mathsf{sigmoid}(dist_k/\sigma), \sigma > 0$, at p_i of the kth buffer. We use $\sigma = \mathbf{r}/7$, unless otherwise noted. To deal with the numerical instability of sigmoid computation, we remove the square for $dist_k^i$ in SoftRas.

Our blending function is generally composed similarly to traditional alpha blending as follows:

$$I(I^e, I^{ne}) = EI^e + (1-E)I^{ne}. \quad (4)$$

where I denotes a pixel value at p_i, and E is a binary edge mask. I^e and I^{ne} are the blended pixel values for the edges and non-edges, respectively. Our formulation explicitly distinguishes blending for edges, which involve many vertical surfaces from a view and have a strong signal for optimization. Thus, I^e should have transparency and far-range gradients. In contrast, non-edges have a weak signal, and transparency for I^{ne} is not necessary in most cases. We compute E from S^H, which is a stencil mask of hard faces in the frontal buffer by edge detection and dilation. Therefore, its width, $\delta \leq \mathbf{r}$, of E is adjustable. We set $\delta = \mathbf{r}$, unless otherwise noted.

To compute a color pixel value, I_c, we use weighted averaging for edges, I_c^e, and keep the most frontal shaded color, C_1, for non-edges, I_c^{ne}.

$$I_c = I(I_c^e, I_c^{ne}),\ I_c^e = \frac{\sum_k \mathcal{D}_k C_k}{\sum_k \mathcal{D}_k},\ I_c^{ne} = S^H C_1. \quad (5)$$

where C_k denotes the shaded color of the kth buffer. We do not use depth values for our color blending as SoftRas because transparency tuning is difficult.

For a silhouette pixel value, I_s, we use binary occupancy blending for edges, I_s^e, and flat intensity for non-edges, I_s^{ne}.

$$I_s = I(I_s^e, I_s^{ne}),\ I_s^e = 1 - \prod_k (1-\mathcal{D}_k),\ I_s^{ne} = S^H. \quad (6)$$

Our formulation above generalizes and bridges anti-aliasing and SoftRas. A comparison of HardSoftRas with some settings and

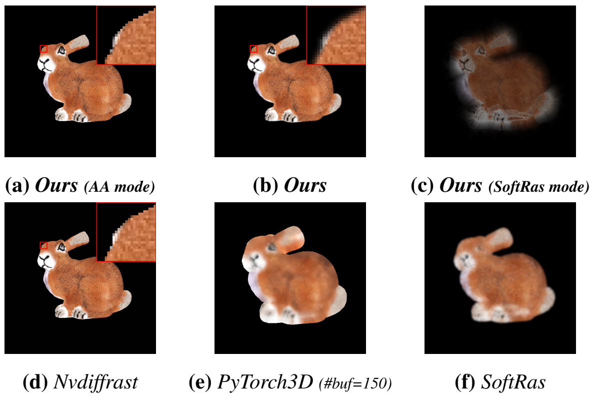


Figure 7: *Comparison of HardSoftRas with several settings and existing methods with bunny model (#face=5k) of image size 512 × 512. We use* $\mathbf{r} = 0.1, \sigma = 0.0285$ *for (c) and set the corresponding parameters to (e) and (f). Red rectangles for (a), (b), and (d) show blurring range differences.*

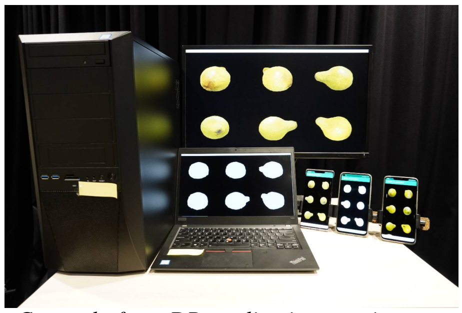


Figure 8: *Cross-platform DR application running on various devices: a desktop PC with NVIDIA, laptop with Intel, and smartphones with Arm.*

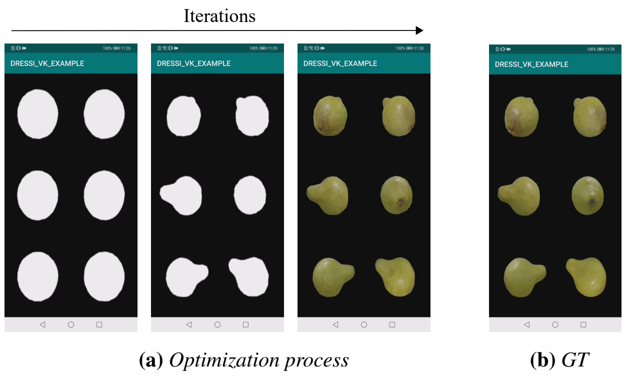


Figure 9: *Cross-platform DR application on a mobile platform. It optimizes the vertex positions and albedo texture of a white sphere to fit the rendered result to the green pear images. (a) Optimization process. From left to right: rendered images with an initial sphere geometry with a white albedo texture, optimized geometry, and optimized geometry and texture. (b) Rendered GT pear model.*

the existing DR techniques is shown in Fig. 7. For example, Nvdiffrast is inspired by distance-to-edge AA [Mal18] and geometric post-process AA [Emi11]. It detects edges in the screen space geometrically, computes pixel-to-edge distances for pixels closer to the edges of less than one pixel in the most frontal buffer, and blends colors for them. When we set δ to one pixel equivalent with $\mathbf{K} = 1$, our method can approximate Nvdiffrast with a subtle visual difference ((a) and (d)). One difference is that we blur all edge pixels, whereas Nvdiffrast selects a part of the edge pixels to be blended. Moreover, our edge detection employs image processing for silhouette edges, but Nvdiffrast's geometric method may detect inside edges on depth discontinuity. On the other hand, SoftRas and PyTorch3D blend the non-edge region besides the edge region. By setting $E(p_i) = 1 | p_i \in \forall i$, our formulation approaches them by blending faces inside silhouettes ((c), (e), and (f)). Our appearance is slightly different, owing to the updated blending function and edge-pixel distance computing region of the face-wise approach. Our moderate and preferred setting (b) with $\mathbf{r} = 0.01, \mathbf{K} = 3$ maintains a sharp texture inside silhouettes and generates far-range gradients around the edges. In general, far-range settings like SoftRas converge faster, while near-range settings like AA are useful to accurately optimize complex scenes with many overlaps. A recent optimization technique [NJJ21] can be combined with our method.

4. Validation

4.1. Cross-Platform Validation

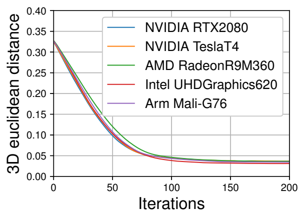
To demonstrate the hardware independence, we ran the same geometry and texture optimization application on four major platforms: NVIDIA for workstations and servers, AMD for desktops, Intel for laptop PCs, and Arm for smartphones. Fig. 8 depicts the application running on various hardware. The application optimizes an initial sphere with a white albedo texture, bringing images rendered in six views closer to synthetic pear images. Rendering image size per view and albedo texture size are 256 × 256. First, the application optimizes the vertex positions by silhouettes with the L2 loss and Laplacian regularization. After the geometry is aligned to the pear silhouettes, we optimize the diffuse albedo texture, minimizing color L2 loss. We use the Adam optimizer for both optimization processes, and the camera parameters are fixed as the ground truth. The same parameters are used for all the platforms. Fig. 9 shows screenshots of the application on a mobile platform with Arm.

Table 2 shows our detailed experimental settings, stage packing effects and performance. The maximum number of input attachments for each hardware type, obtained from the Vulkan API, is a constraint used for stage packing. Although more constraints are considered, we omit other constraints for need of space. Our stage packing faithfully generates optimized stages for each platform: fewer stages for less constraint hardware (i.e., NVIDIA and AMD). Our reactive cache reduces more stages for texture optimization by skipping constant rasterization results. The two rightmost columns are the average optimization time per iteration of the platforms. NVIDIA RTX2080 for gaming workstations shows the best number. TeslaT4 for deep learning servers is slightly slower than the RTX2080 because it is not optimized for graphics purposes. The second is an Intel for laptops, the third is an AMD for office desktops, and the last is an Arm for smartphones. The stages reduced by our reactive cache make texture optimization faster than geometry optimization in all the platforms.

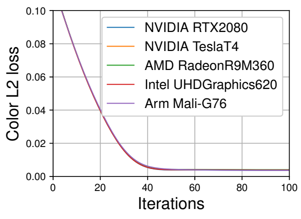
Fig. 10 shows the error curves against GT. We take the averages of over 10 trials to handle the non-deterministic behaviors

					#Substage / #Stage (Before → After cache)		Time per iter. (ms)	
Vendor	Model	Usage	Used OS	Maximum #input attach.	Geo. opt.	Tex. opt.	Geo. opt.	Tex. opt.
NVIDIA	RTX2080	Workstation	Windows	1048576	218/129 → 206/122	187/160 → 86/69	3.0	2.0
	TelsaT4	Server	Ubuntu	1048576	220/131 → 207/124	186/160 → 88/70	6.1	4.1
AMD	RadeonR9M360	Desktop	Windows	N.A.	219/129 → 206/122	187/160 → 86/69	51.4	28.8
Intel	UHDGraphics620	Laptop	Windows	8	348/134 → 336/128	212/181 → 111/84	38.3	23.5
Arm	Mali-G76	Smartphone	Android	4	687/142 → 670/132	278/187 → 192/86	420.0	189.5

Table 2: *Stage packing and performance evaluation of the cross-platform DR application on various hardware. As an example of the hardware constraints for our stage packing, we show the maximum number of input attachments taken from the Vulkan API. The larger the value of the input attachment is, the fewer the number of stages, and the more efficient the execution. The hardware constraint is a concept orthogonal to the hardware performance (e.g., clock). The time taken for optimization depends on both hardware constraints and performance.*



(a) *Geometry optimization*



(b) *Texture optimization*

Figure 10: *Error against GT on several platforms. Errors converge to almost the same value for all platforms, which indicates that our method is hardware-agnostic. (a) 3D distance (not used as a loss) between GT and optimized geometry in geometry optimization. (b) Color L2 loss during texture optimization.*

Optimization target:	Vertex position	Albedo texture
Ours full	**58.441**	**30.874**
-stage pack	110.686	47.462
-reactive cache	317.814	306.594
-stage pack, reactive cache	364.524	315.741
baseline	1780.90	1652.14

Table 3: *Performance validation for stage packing and reactive cache. The average time (ms) of 1,000 iterations to optimize vertex positions and albedo texture on Arm Mali-G76. Avocado geometry is optimized with a single view. Ours full on the top is combined with stage pack, substage pack and reactive cache. Lower lines with "-" remove individual components. The baseline is naive shader implementation, and "-stage pack, reactive cache" (substage pack only) is a similar case to traditional kernel fusion technique.*

of GPUs. For geometry optimization, we show 3D distance error curves against GT geometry. They converge at approximately 150 iterations on all platforms. Converged error values are very similar for all platforms. For texture optimization, all platforms converge around 50 iterations. The AMD shows larger values than others in the middle of geometry optimization. Although graphics pipeline API is common, its implementation (e.g., floating-point precision) depends on vendors. Therefore, optimization results may have a little numerical difference among hardware.

4.2. Stage Packing Validation

We performed an ablation study to validate that our stage packing and reactive cache contribute to the fast speed. As a baseline of stage packing off, we used a naive method to assign each function object to a single stage without the greedy strategy.

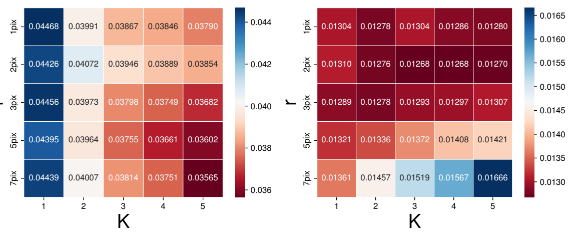
We performed the validation on Arm Mali-G76 for two reasons. First, it is designed for mobile devices and has limited hardware capability. Dressi should accelerate the DR on such weak hardware. Second, it is based on a tile-based GPU architecture and suitable for confirming speed improvement by substages and the corresponding subpasses. A subpass in Vulkan is designed for tile-based rendering, and its proper utilization reduces bandwidth use [addoc21]. Table 3 shows the comparison with several settings. Our proposed approaches successfully accelerate forward and backward rendering speeds.

4.3. HardSoftRas Validation

We analyze our optimization capability with several sets of hyperparameters. We optimize vertex positions of a coarse sphere geometry to make it visually similar to a dense synthetic GT model, which is a skull geometry with a complex and uneven shape. At each iteration, a random view with a random point light in Lambertian shading is rendered. The L1 loss for the rendered GT with the same configuration is minimized with the scheduled Laplacian regularization term. We ran 2,500 iterations in 256×256 image resolution. As a reference, we compared ours with Nvdiffrast, using the same parameters and settings. We ran this experiment on a desktop PC with NVIDIA RTX2080, and the average 3D distance of vertices to the closest GT surface was used as an evaluation metric.

Fig. 11 shows the validation as a heatmap with **K** and **r** combinations. At 500 iterations (a), wide blurring settings with large **K** and **r** show better results. In contrast, at 2,500 iterations (b), a small **r** yields a better convergence. The results show that far-range gradients are preferred at the early stages of optimization because the object being optimized is far from the target. However, at the latter stages of the process, it is better to use near-range gradients to refine the details.

Next, we compare the error curves of our method and Nvdiffrast in Fig. 12. For our method, we plotted three configurations



(a) *At iter. 500* **(b)** *At iter. 2,500*

Figure 11: *Validation heatmaps for hyper parameters. The 3D distances against the GT with several* **K** *(horizontal) and* **r** *(vertical) combinations are shown. The upper left with small* **r** *and* **K** *is a near-range gradient setting, and the lower right with large* **r** *and* **K** *is for far-range gradients. (a) After 500 iterations. Settings with larger* **r** *and larger* **K** *show better distances. (b) After 2,500 iterations. Settings with smaller* **r** *show better distances.*

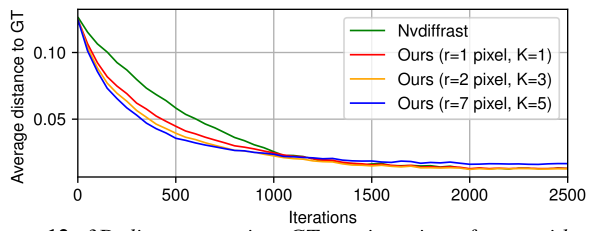


Figure 12: *3D distance against GT per iteration of ours with selected parameters and Nvdiffrast. Ours shows similar tendency to the heatmap and shows faster convergence than Nvdiffrast.*

based on the validation heatmaps: (1) the best parameter at iteration 2,500 ($\mathbf{r} = 2\text{pix}, \mathbf{K} = 3$), (2) parameters for a very far-range gradient ($\mathbf{r} = 7\text{pix}, \mathbf{K} = 5$), and (3) parameters for a very near-range gradient ($\mathbf{r} = 1\text{pix}, \mathbf{K} = 1$). The error curves show the same tendency as the heatmaps. Our convergence is faster in all the settings than Nvdiffrast. Regarding accuracy, the final distance in 2,500 iterations, excluding the very far-range setting, is better than that of Nvdiffrast.

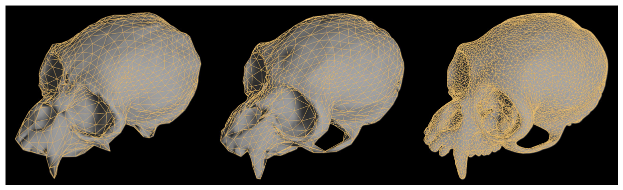
Finally, we apply normal map optimization to the optimized geometries to refine the appearance closer to the GT. We used almost the same setting for geometry optimization but increased the size of rendered images to 1024×1024. The size of a normal map was also set to 1024×1024. We visually compare our method with the configuration (1) and Nvdiffrast in Fig. 13. Our method shows better convergence to the GT with a closed ring shape on the side.

5. Experiments

In this section, we evaluate our Dressi by comparing existing methods. All experiments were run on a single NVIDIA GeForce RTX 2080, Intel(R) Core(TM) i7-9700K, and 32GB RAM.

5.1. Performance Comparison

To validate that the performance of our renderer is robust against several parameters, we measured the computational costs to calculate forward and backward functions in two settings. We compare



(a) *Nvdiffrast* **(b)** ***Ours*** **(c)** *GT*

Figure 13: *Visual comparison of optimized results with wire frame. (a) Nvdiffrast, (b) Ours, and (c) GT*

	#Vertices	406	7107	25697
	#Triangles	682	14208	51414
	Window res.	Fwd + bwd time per frame (ms)		
Ours	256x256	0.304	0.308	0.364
	512x512	0.442	0.480	0.502
	1024x1024	1.034	1.106	1.104
	2048x2048	3.301	3.545	3.469
Nvdiffrast [LHK*20]	256x256	1.060	0.480	0.646
	512x512	1.305	0.929	0.853
	1024x1024	2.371	2.523	2.086
	2048x2048	5.403	8.010	5.518
PyTorch3D [RRN*20]	256x256	8.054	10.272	9.393
	512x512	12.570	15.354	14.919
	1024x1024	32.136	39.034	36.382
	2048x2048	105.823	134.795	121.624

Table 4: *Performance comparison with non-textured meshes to optimize vertex positions. Ours and Nvdiffrast render silhouettes. For PyTorch3D, we apply Gouraud shading.*

Dressi with Nvdiffrast and PyTorch3D. Because none of the methods supported the same shaders, we used shaders as similar as possible. We show the average rendering time of 1,000 frames, removing outliers outside the 95% confidence interval for all recorded data.

First, we evaluated the performance against the number of triangles and the resolution of window sizes. To consider the effect of view-dependent geometry complexity, we rendered the meshes in six views with white vertex color and averaged the time spent on rendering. Table 4 shows the performance of our method outperforms those of Nvdiffrast and PyTorch3D for all meshes and window resolutions.

Next, we demonstrate the scalability against the texture resolutions with shaders that map diffuse albedo textures [Dem21] to the meshes. We changed the texture resolutions for each mesh with the fixed window resolution of 2048×2048. Table 5 shows that our performance is at least twice as fast as the others for all texture resolutions.

5.2. 3D Reconstruction with Real Data

We also performed practical 3D reconstruction with Dressi and Nvdiffrast. We used a real dataset for multi-view object reconstruction, DTU dataset [AJV*16], and its supplemental silhouettes

	#Vertices	31027	7232
	#Triangles	55704	14217
	Texture res.	Fwd + bwd time per frame (ms)	
Ours	1024x1024	2.929	2.893
	2048x2048	3.965	3.953
	4096x4096	8.148	8.045
Nvdiffrast [LHK*20]	1024x1024	6.584	6.254
	2048x2048	8.922	8.653
	4096x4096	18.676	18.635
PyTorch3D [RRN*20]	1024x1024	137.681	118.184
	2048x2048	143.237	119.267
	4096x4096	141.667	120.017

Table 5: *Performance comparison with textured-meshes to optimize diffuse albedo textures. Ours and Nvdiffrast render color images with unlit shading. We apply Phong shading for PyTorch3D.*

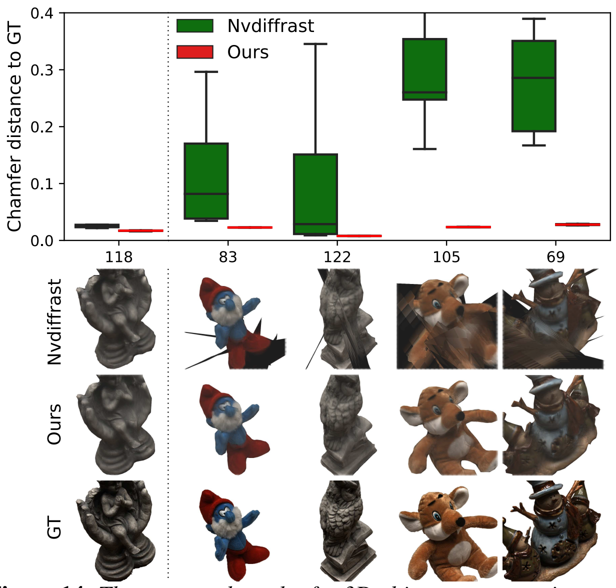
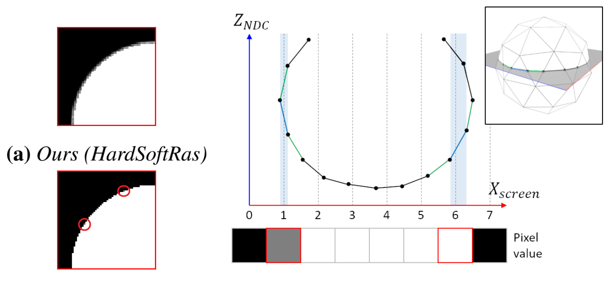


Figure 14: *The top row: box plot for 3D object reconstruction task with real data. Statistics with 10 trials of Chamfer distance are shown vertically, and the horizontal numbers are object IDs. Ours shows smaller variances and better distances for all objects. The bottom row: visual comparison with a reference GT image. We visualize optimized models corresponding to median Chamfer loss. Nvdiffrast and ours are rendered by GT camera parameters.*

[YKM*20]. A big difference from the HardSoftRas validation is the limited number of views. Only 49 or 64 views are available for each object. Another challenge is the color discrepancy between the views caused by the lack of perfect control over the camera settings. We fix the camera parameters as GT and minimize the L2 loss of color and silhouettes in 200 × 150 resolution with unlit shading using the Adam optimizer. Vertex positions and diffuse albedo texture are jointly optimized. We start the optimization with a white sphere, and a Laplacian regularizer with scheduling is used.

The top row of Fig. 14 is a box plot with five real objects for our method and Nvdiffrast with 10 trials to consider non-determinism of GPUs. The chamfer distance (lower is better) was used as the evaluation metric in this experiment. First, we tuned the hyperparameters (learning rate and regularization factors) for Nvdiffrast with an object called "118". Then other four objects were optimized with the same hyperparameters. Our method used the common hyperparameters tuned for Nvdiffrast and $\mathbf{r} = 2\text{pix}, \mathbf{K} = 1$ for all five objects. During optimization, we sometimes observed sudden shape collapse for Nvdiffrast, which is reflected as a large variance in the box plot. In contrast, ours never suffers such collapse with slight variance among the trials.



(a) *Ours (HardSoftRas)*

(b) *Nvdiffrast* **(c)** *Geometric anti-aliasing of Nvdiffrast*

Figure 15: *Sphere rendering comparison. (a) Our HardSoftRas blurs all edge pixels. (b) Nvdiffrast updates only two edge pixels in red circles. (c) How the geometric AA of Nvdiffrast works on a white curved surface with a black background. We show a slice of a simplified sphere in the upper right. Geometric edges are in continuous normalized device coordinate (NDC) space and are projected onto discrete pixels. Red pixels are target edge pixels, and the corresponding blue geometric edges are located on the pixel center and have the closest depth. See left red rectangle pixel affected by AA. Blue edge occludes a neighboring green edge at the pixel center. Thus, its color is blended into gray, and the gradient for the pixel is generated accordingly. In contrast, the pixel in the right red rectangle is not affected by AA because the corresponding blue edge does not occlude a neighboring green edge.*

To analyze these behaviors, we show the initial white sphere rendered by each method and how the AA of Nvdiffrast works in Fig. 15. Although (a) ours shows uniformly blurred edges, (b) Nvdiffrast selectively blurs the edge pixels. As shown in (c), Nvdiffrast follows conventional geometric anti-aliasing criteria to select a limited number of edge pixels for color blending and gradient generation. Therefore, the gradients of Nvdiffrast in the screen space are sparse and discontinuous, and optimization using it is unstable. Our HardSoftRas, which blends all edge pixels, ensures dense and smooth far-range gradients, achieving more robust optimization.

The bottom row of Fig. 14 shows a visual comparison of the reconstruction results with the median accuracy and GT. The results of Nvdiffrast suffer from shape artifacts. Our method can successfully reconstruct real objects with complex high-curvature surfaces.

6. Applications

The proposed differentiable renderer, Dressi, has the potential to support many applications. We demonstrate that Dressi can ren-

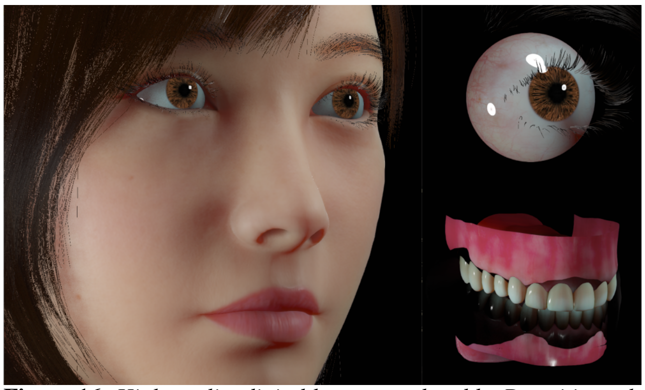


Figure 16: *High quality digital human rendered by Dressi in real-time. It contains skin, hair, eyeball, teeth shaders, and shadowing.*

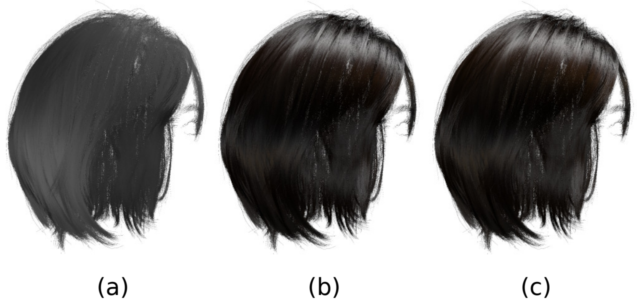


Figure 17: *Hair optimization: (a) rendered with initial parameter, (b) appearance after optimization of biophysical parameters, and (c) target image of the optimization. In the biophysical parameters, melanin components (melanin and melanin redness) affect hair color, and roughness parameters (radial and azimuthal roughness) control the highlight size, and IOR is for the highlight position. The parameters are shared over all strands. We optimized the parameters under the constraints of melanin and roughness from 0.0 to 1.0 and IOR in the range of from 1.0 to 2.0.*

der a photorealistic facial model with complex shaders for human skin, eyeballs, and hair materials. The supplemental material shows more applications: environment map optimization, normal map optimization and human face modeling with 3D morphable model.

6.1. Practical Graphics Rendering with Complex Shaders

We show that our system can support practical complex shaders, including hair, skin, eyeballs, and teeth shaders like recent game engines [Epi21; Uni21]. Fig. 16 shows the rendering results of our high-quality digital human model with them. Moreover, our engine supports the backward pass for the complex shaders.

To demonstrate inverse rendering with the complex shaders, we performed optimization of hair shading with the Marschner shader [MJC*03], which is a well-known physically-based reflection model for hair materials. Biophysical parameters, including melanin, melanin redness, radial and azimuthal roughness, and index of refraction (IOR), were optimized. The geometry and camera parameters were known, and an image from only one viewpoint was used. Fig. 17 shows (a) the initial, (b) optimized, and (c) target images. The optimized image (b) has the same appearance as the target image (c).

7. Conclusion and Future Work

In this paper, we proposed Dressi, a hardware-agnostic and highly efficient rasterization-based differentiable renderer to support various graphics hardware. Dressi is based on a new design in which DR algorithms are completely written by AD for DR. Our Dressi-AD realizes hardware independence by Vulkan API and an inverse UV technique. Dressi-AD employs stage packing, a runtime optimization method with a reactive cache to accelerate computational graph executions and adapt hardware constraints. HardSoftRas, our proposed rendering process entirely implemented on Dressi-AD, is the first DR algorithm to generate screen space far-range gradients under the limitation of graphics pipelines. We demonstrated that our approach successfully works on a variety of graphics hardware (i.e., NVIDIA, AMD, Intel, and Arm). We validated that our stage packing is adaptive to hardware constraints and contributes to speed. Our Dressi has the potential to realize a wide range of practical applications in various devices. Our experiments with synthetic and real data demonstrated that the speed and quality of our renderer outperform those of other state-of-the-art methods. Our rendering for a digital human is of high quality and can run backward passes. We show the optimization of hair color, which is not supported by other rasterization-based DR frameworks.

We leave two research topics for more practical problems in our framework. The first topic is about the propagation of gradients of vertex attributes. HardSoftRas employs depth peeling to consider gradients of triangles occluded in screen space. However, the computational cost increases in proportion to many z-buffers. Although we prioritize important triangles to reduce the number of z-buffers, other order-independent transparency methods may realize more efficient gradient propagation. The second topic is the further flexibility of our renderer. Combinations of DR and neural networks are often used, such as optimization with identity loss [GPKZ19]. We do not introduce operators of neural networks into our framework. Connecting common AD libraries through the CUDA interface is possible, but this approach sacrifices hardware independence. We can define the standard operators of neural networks in basic operations of GLSL functions, and they provide more flexible solutions for practical problems. We believe that Dressi contributes to the further improvement of rasterization-based DR, including the solutions of the referred two topics.

Acknowledgments

We thank the anonymous reviewers for their constructive comments, which have helped us improve the manuscript. We also want to thank Toshiya Hachisuka for helpful discussions. We are grateful to Naoya Iwamoto for designing visual materials. We appreciate Morales Espinoza Carlos Emanuel and Zhengqing Li for discussing related work and their thoughtful comments. We thank Naoya Hirai and Chun Geng for the appearance of skin shading. We are also grateful to Baris Gecer, Athanasios Vogiannou, Stylianos Moschoglou, Stylianos Ploumpis, Alexandros Lattas, and Stefanos Zafeiriou for the evaluation of GANFit [GPKZ19] and GANFit++ [GPKZ21].

References

[addoc21] Arm developer DOCUMENTATION. *Tile-Based Rendering*. 2021. URL: https://developer.arm.com/documentation/102662/latest/ 10.

[AJV*16] AANÆS, HENRIK, JENSEN, RASMUS RAMSBØL, VOGIATZIS, GEORGE, et al. "Large-Scale Data for Multiple-View Stereopsis". *International Journal of Computer Vision* (2016), 1–16 11.

[ALKN19] AZINOVIC, DEJAN, LI, TZU-MAO, KAPLANYAN, ANTON, and NIEßNER, MATTHIAS. "Inverse Path Tracing for Joint Material and Lighting Estimation". *CVPR*. 2019 1.

[AM*10] AGARWAL, SAMEER, MIERLE, KEIR, et al. *Ceres Solver*. 2010. URL: http://ceres-solver.org 3.

[BCC*13] BAINOMUGISHA, ENGINEER, CARRETON, ANDONI LOMBIDE, CUTSEM, TOM VAN, et al. "A Survey on Reactive Programming". *ACM Computing Surveys* 45.4 (2013) 6.

[CGL*19] CHEN, WENZHENG, GAO, JUN, LING, HUAN, et al. "Learning to Predict 3D Objects with an Interpolation-based Differentiable Renderer". *NIPS*. 2019 1, 3.

[CNS*02] CHAN, ERIC, NG, REN, SEN, PRADEEP, et al. *Efficient partitioning of fragment shaders for multipass rendering on programmable graphics hardware*. Tech. rep. STANFORD UNIV CA, 2002 6.

[Dem21] DEMES, LENNART. *CC0 TEXTURES*. 2021. URL: https://cc0textures.com/ 11.

[DMZ*17] DEVITO, ZACHARY, MARA, MICHAEL, ZOLLHÖFER, MICHAEL, et al. "Opt: A Domain Specific Language for Non-Linear Least Squares Optimization in Graphics and Imaging". *ACM Trans. Graph.* 36.5 (2017). ISSN: 0730-0301 3.

[Emi11] EMIL, PERSSON. *Geometric Post-Process Anti-Aliasing*. 2011. URL: https://www.humus.name/index.php?page=3D&ID=86 9.

[Epi21] EPIC GAMES INC. *Unreal Engine*. 2021. URL: https://www.unrealengine.com/ 13.

[EST*20] EGGER, BERNHARD, SMITH, WILLIAM A. P., TEWARI, AYUSH, et al. "3D Morphable Face Models - Past, Present, and Future". *ACM Trans. Graph.* 39.5 (2020). ISSN: 0730-0301 1.

[Eve01] EVERITT, CASS. "Interactive order-independent transparency". *White paper, nVIDIA* 2.6 (2001), 7 3.

[Fuj08] FUJITA, SYOYO. *MUDA (MUltiple Data Accelerator) language compiler*. 2008. URL: https://github.com/syoyo/mudalang 3.

[GPKZ19] GECER, BARIS, PLOUMPIS, STYLIANOS, KOTSIA, IRENE, and ZAFEIRIOU, STEFANOS. "Ganfit: Generative adversarial network fitting for high fidelity 3d face reconstruction". *CVPR*. 2019 1, 13.

[GPKZ21] GECER, BARIS, PLOUMPIS, STYLIANOS, KOTSIA, IRENE, and ZAFEIRIOU, STEFANOS. "Fast-GANFIT: Generative Adversarial Network for High Fidelity 3D Face Reconstruction". *IEEE Transactions on Pattern Analysis & Machine Intelligence* 01 (May 2021), 1–1. ISSN: 1939-3539. DOI: 10.1109/TPAMI.2021.3084524 1, 13.

[GRF11] GUENTER, BRIAN, RAPP, JOHN, and FINCH, MARK. *Symbolic differentiation in GPU shaders*. Tech. rep. Citeseer, 2011 3.

[Gri92] GRIEWANK, ANDREAS. "Achieving Logarithmic Growth Of Temporal And Spatial Complexity In Reverse Automatic Differentiation". *Optimization Methods and Software* 1.1 (1992) 3.

[GW08] GRIEWANK, ANDREAS and WALTHER, ANDREA. *Evaluating Derivatives*. Society for Industrial and Applied Mathematics, 2008 3.

[HAL*20] HU, YUANMING, ANDERSON, LUKE, LI, TZU-MAO, et al. "DiffTaichi: Differentiable Programming for Physical Simulation". *ICLR* (2020) 3.

[HAO05] HASSELGREN, JON, AKENINE-MÖLLER, TOMAS, and OHLSSON, LENNART. "Conservative rasterization". *GPU Gems 2*. Addison-Wesley, 2005, 677–690 7.

[HFF18] HE, YONG, FATAHALIAN, KAYVON, and FOLEY, TIM. "Slang: language mechanisms for extensible real-time shading systems". *ACM Transactions on Graphics (TOG)* 37.4 (2018), 1–13 3.

[HML*21] HASSELGREN, JON, MUNKBERG, JACOB, LEHTINEN, JAAKKO, et al. "Appearance-Driven Automatic 3D Model Simplification". *Eurographics Symposium on Rendering*. 2021 3.

[HXKD20] HU, YUANMING, XU, MINGKUAN, KUANG, YE, and DURAND, FRÉDO. "AsyncTaichi: Whole-Program Optimizations for Megakernel Sparse Computation and Differentiable Programming". (2020). arXiv: 2012.08141 [cs.PL] 3.

[Kaj86] KAJIYA, JAMES T. "The Rendering Equation". *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '86. New York, NY, USA: ACM, 1986, 143–150. ISBN: 0-89791-196-2 2.

[KBM*20] KATO, HIROHARU, BEKER, DENIZ, MORARIU, MIHAI, et al. "Differentiable Rendering: A Survey". (2020). arXiv: 2006.12057 [cs.CV] 3.

[Khr21] KHRONOS GROUP. *Vulkan*. 2021. URL: https://www.khronos.org/vulkan/ 1, 2.

[KUH18] KATO, HIROHARU, USHIKU, YOSHITAKA, and HARADA, TATSUYA. "Neural 3D Mesh Renderer". *CVPR*. 2018 1, 3.

[LADL18] LI, TZU-MAO, AITTALA, MIIKA, DURAND, FRÉDO, and LEHTINEN, JAAKKO. "Differentiable Monte Carlo Ray Tracing through Edge Sampling". *ACM Trans. Graph.* 37.6 (2018). ISSN: 0730-0301 1, 2.

[Lat08] LATTNER, CHRIS. "LLVM and Clang: Next generation compiler technology". *The BSD conference*. Vol. 5. 2008 3.

[LB14] LOPER, MATTHEW M. and BLACK, MICHAEL J. "OpenDR: An Approximate Differentiable Renderer". *ECCV*. 2014 3.

[LHK*20] LAINE, SAMULI, HELLSTEN, JANNE, KARRAS, TERO, et al. "Modular Primitives for High-Performance Differentiable Rendering". *ACM Trans. Graph.* 39.6 (2020). ISSN: 0730-0301 2–5, 11, 12.

[LLCL19] LIU, SHICHEN, LI, TIANYE, CHEN, WEIKAI, and LI, HAO. "Soft Rasterizer: A Differentiable Renderer for Image-based 3D Reasoning". *ICCV*. 2019 2–4.

[LLL*21] LI, MINGZHEN, LIU, YI, LIU, XIAOYAN, et al. "The Deep Learning Compiler: A Comprehensive Survey". *IEEE Transactions on Parallel and Distributed Systems* 32.3 (2021), 708–727 3.

[Mal18] MALAN, HUGH. "Edge Antialiasing by Post-Processing". (2018), 33–58 9.

[MAP*15] MARTÍN ABADI, ASHISH AGARWAL, PAUL BARHAM, et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. URL: http://tensorflow.org/ 3.

[MC20] MOSES, WILLIAM and CHURAVY, VALENTIN. "Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients". *Advances in Neural Information Processing Systems*. Ed. by LAROCHELLE, H., RANZATO, M., HADSELL, R., et al. Vol. 33. Curran Associates, Inc., 2020, 12472–12485. URL: https://proceedings.neurips.cc/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf 3.

[MGAK03] MARK, WILLIAM R, GLANVILLE, R STEVEN, AKELEY, KURT, and KILGARD, MARK J. "Cg: A system for programming graphics hardware in a C-like language". *ACM SIGGRAPH 2003 Papers*. 2003, 896–907 5.

[MJC*03] MARSCHNER, STEPHEN R., JENSEN, HENRIK WANN, CAMMARANO, MIKE, et al. "Light scattering from human hair fibers". *ACM Trans. Graph.* 22.3 (2003) 13.

[MSPK06] MCGUIRE, MORGAN, STATHIS, GEORGE, PFISTER, HANSPETER, and KRISHNAMURTHI, SHRIRAM. "Abstract shade trees". *Proceedings of the 2006 symposium on Interactive 3D graphics and games*. 2006, 79–86 6.

[Mur12] MURANUSHI, TAKAYUKI. "Paraiso: An Automated Tuning Framework for Explicit Solvers of Partial Differential Equations". *Computational Science & Discovery* 5.1 (2012), 015003 3.

[NDJK21] NIMIER-DAVID, MERLIN, DONG, ZHAO, JAKOB, WENZEL, and KAPLANYAN, ANTON. "Material and Lighting Reconstruction for Complex Indoor Scenes with Texture-space Differentiable Rendering". (2021) 5.

[NJJ21] NICOLET, BAPTISTE, JACOBSON, ALEC, and JAKOB, WENZEL. "Large Steps in Inverse Rendering of Geometry". *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)* 40.6 (Dec. 2021). DOI: 10.1145/3478513.3480501 9.

[NSRJ20] NIMIER-DAVID, MERLIN, SPEIERER, SÉBASTIEN, RUIZ, BENOÎT, and JAKOB, WENZEL. "Radiative Backpropagation: An Adjoint Method for Lightning-Fast Differentiable Rendering". *ACM Trans. Graph.* 39.4 (2020). ISSN: 0730-0301 2.

[NVZJ19] NIMIER-DAVID, MERLIN, VICINI, DELIO, ZELTNER, TIZIAN, and JAKOB, WENZEL. "Mitsuba 2: A Retargetable Forward and Inverse Renderer". *ACM Trans. Graph.* 38.6 (2019). ISSN: 0730-0301 1–3.

[ODo17] O'DONNELL, YURIY. "FrameGraph: Extensible Rendering Architecture in Frostbite". *GDC*. 2017 6.

[OUN*17] OKUTA, RYOSUKE, UNNO, YUYA, NISHINO, DAISUKE, et al. "CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations". *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*. 2017 3.

[PGM*19] PASZKE, ADAM, GROSS, SAM, MASSA, FRANCISCO, et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". *Advances in Neural Information Processing Systems 32*. Curran Associates Inc., 2019, 8024–8035 3.

[RRN*20] RAVI, NIKHILA, REIZENSTEIN, JEREMY, NOVOTNY, DAVID, et al. "Accelerating 3D Deep Learning with PyTorch3D". (2020). arXiv: 2007.08501 [cs.CV] 3–5, 11, 12.

[RRR*15] RHODIN, HELGE, ROBERTINI, NADIA, RICHARDT, CHRISTIAN, et al. "A Versatile Scene Model with Differentiable Visibility Applied to Generative Pose Estimation". *Proceedings of the 2015 International Conference on Computer Vision (ICCV 2015)*. 2015 1.

[Sob67] SOBOL', I.M. "On the distribution of points in a cube and the approximate evaluation of integrals". *USSR Computational Mathematics and Mathematical Physics* 7.4 (1967), 86–112. ISSN: 0041-5553 5.

[STA*19] SMILKOV, DANIEL, THORAT, NIKHIL, ASSOGBA, YANNICK, et al. "Tensorflow. js: Machine learning for the web and beyond". *Proceedings of Machine Learning and Systems* 1 (2019), 309–321 3, 5.

[Uni21] UNITY TECHNOLOGIES. *Unity*. 2021. URL: https://unity.com/ 13.

[VKP*19] VALENTIN, JULIEN, KESKIN, CEM, PIDLYPENSKYI, PAVEL, et al. "TensorFlow Graphics: Computer Graphics Meets Deep Learning". 2019 1, 3.

[VSJ21] VICINI, DELIO, SPEIERER, SÉBASTIEN, and JAKOB, WENZEL. "Path Replay Backpropagation: Differentiating Light Paths using Constant Memory and Linear Time". *ACM Trans. Graph.* 40.4 (2021), 108:1–108:14 2, 3.

[WLY10] WANG, GUIBIN, LIN, YISONG, and YI, WEI. "Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU". *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*. IEEE. 2010, 344–350 3.

[YKM*20] YARIV, LIOR, KASTEN, YONI, MORAN, DROR, et al. "Multiview Neural Surface Reconstruction by Disentangling Geometry and Appearance". *Advances in Neural Information Processing Systems* 33 (2020) 12.

[ZMY*20] ZHANG, CHENG, MILLER, BAILEY, YAN, KAI, et al. "Path-Space Differentiable Rendering". *ACM Trans. Graph.* 39.4 (2020), 143:1–143:19 2.