UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Master's Thesis

Alex Meislich

3D Gaussian Splatting with Futhark

Using Automatic Differentiation with an Array-Based Functional Language to Implement 3DGS in a Massively Parallel Context

Advisor: Troels Henriksen

Handed in: September 24, 2026

Contents

Chapter 1

Abstract

3D Gaussian Splatting (3DGS) has emerged in recent years as a state-of-the-art technique for inverse rendering and novel view synthesis, but existing implementations rely on manually derived gradient functions that are difficult to write and maintain. This thesis investigates whether Futhark, an array-based functional programming language with native support for automatic differentiation (AD), can serve as a viable replacement for manual differentiation in this setting.

I begin by covering the physics behind volume rendering and 3DGS, discussing prior work in the field, weighing the trade-offs of various approaches to inverse rendering, and offering a technical explanation of the 3DGS algorithm. I then present a purpose-built 3DGS renderer written in Futhark, exposing both naive and optimized reverse-mode AD gradient functions, and integrate it with the PyTorch training loop supplied by the original 3DGS authors. I evaluate this setup on standard benchmark scenes, comparing training time, inference speed, and image quality against the reference CUDA implementation. Ultimately, I find that although AD in Futhark simplifies the implementation—requiring far less code and eliminating the need for manual gradient derivation—its runtime performance suffers greatly. As such, I conclude that automatically derived Futhark functions are not always suitable as drop-in replacements for manually derived gradients in this setting. Instead, I note Futhark's promise as a prototyping tool for developing proofs of concept without the technical hurdles imposed by a low-level language like CUDA.

Chapter 2

Background

2.1 Rendering and Inverse Rendering

Across various scientific and professional fields, there is a need to edit, model, and store localized portions of three-dimensional space. These portions are referred to as *scenes*, and their corresponding digital formats are known as *scene representations*. Because scene representations necessarily encode three-dimensional information, they are fundamentally incompatible with flat computer displays, which are only capable of depicting two-dimensional images. To visualize a scene's representation, therefore, a specialized computer program known as a *renderer* is used to generate a two-dimensional image of the scene from the perspective of a specified *camera pose*, which defines the position and orientation from which the scene is viewed.

While the physical volume a scene depicts varies enormously across fields—an architect may require a scene to depict a floor of a building, whereas a video game developer may need a scene to depict a magical village deep within the rainforest—all scenes have traditionally been generated from within the computer, meaning their contents are either procedurally constructed or deliberately placed by a human author rather than automatically derived from real-life photographs or measurements. For many applications, however, this pipeline is inappropriate because the desired scene already exists in the physical world, making manual reconstruction unnecessarily time-consuming, expensive, and often infeasible. In these cases, it is more useful to reconstruct a real-world scene using *inverse rendering*, where several photographs are used to generate a scene representation from scratch. Unlike in rendering, where a well-defined scene is used to produce images, inverse rendering works in the opposite direction by building a scene from scratch using images, usually of a real-world scene. Unfortunately, inverse rendering is an ill-posed problem because the photo capturing process is inherently destructive: objects in the foreground obstruct the background and literally hide information about the scene from the camera and, by extension, the inverse renderer.

To overcome the information loss associated with photography, successful approaches to scene reconstruction utilize dozens or hundreds of ground truth images of the target scene, each associated with a camera pose. These ground truth images are used to iteratively update a vector of *scene parameters*, θ, which define the scene representation. This optimization process is typically performed with stochastic gradient descent, which requires computing the gradient of a loss function with respect to θ. Consequently, each optimization iteration of the gradient descent requires a renderer whose output pixels are differentiable with respect to its input parameters. The design of this *differentiable renderer* greatly influences the speed, image quality, and overall performance of the inverse rendering pipeline[13]. At each iteration, the differentiable renderer's output is compared to the ground truth image for the corresponding camera pose. This comparison is performed using a loss function whose gradient is propagated backward through the renderer to update θ. As training (or optimization) progresses, the loss converges, and the scene represented by θ begins to match every ground truth image when rendered from its associated camera pose. The hope is that this training generalizes well enough to support *novel view synthesis*, where the renderer can depict the scene from camera poses that are not present in the ground truth dataset. For a visual depiction of this process, see figure 2.1.

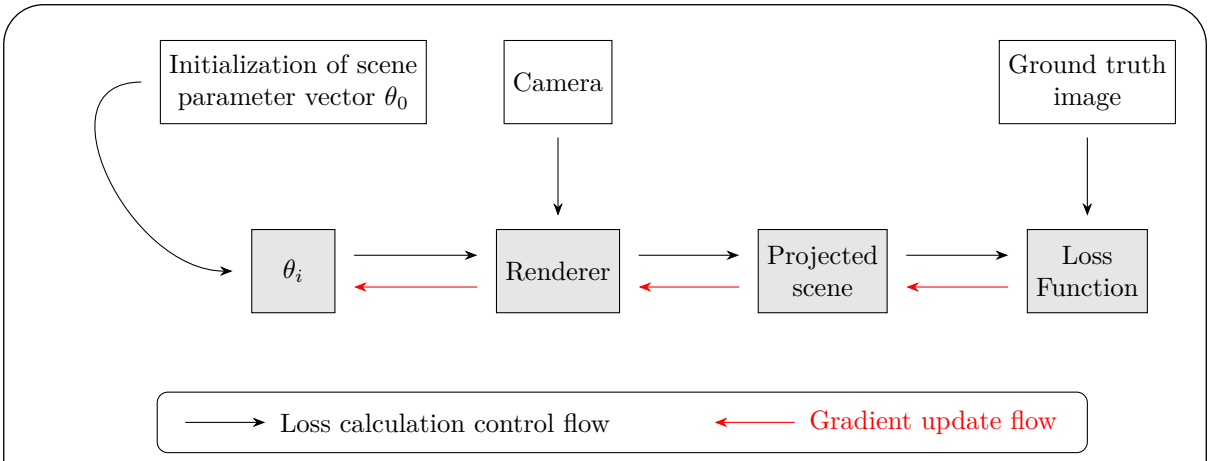

Figure 2.1: **Inverse rendering control flow.** After the scene parameters are initialized, they are passed to the renderer along with a camera. The renderer returns a projection which is fed into a loss function along with the ground truth image associated with the camera used. The derivative of the loss function is then propagated backward, the scene parameters are updated according to some optimizer (not shown, but likely Adam[15]), and the process repeats in the next iteration.

2.2 Optics

The power transmitted by light sources, often referred to as *flux*, is measured by physicists in numerous ways. For the purposes of computer graphics, *radiance* is a particularly useful metric due to its unique property of remaining constant along individual rays of light traveling through empty space[1].

The study of light in a vacuum is only marginally interesting, however, and mostly serves as a stepping stone to understanding its behavior in more commonly appearing media. In particular, we are concerned with how the radiance at a given wavelength, λ, is affected along a ray as it traverses a *particle field* such as a cloud of water vapor, a dust storm, or even a dense medium like a brick wall. Our end goal will be to calculate the final radiance of the ray after it completes its passage through a medium.

When light propagates through a particle field, its radiance along any ray is governed by two competing processes: *emission*, in which particles contribute radiance, and *absorption*, in which particles attenuate it[24, 18]. As the ray traverses the medium, each infinitesimal segment along the ray may add radiance through emission and attenuate it through absorption[2]. This intuition leads us directly to the emission-absorption differential equation[18]

$$\frac{dC_\lambda(q)}{dq} = g(q) - \tau(q)C_\lambda(q) \tag{2.1}$$

where $C_\lambda(q)$ is the radiance at wavelength λ along the ray, q represents distance in the direction of light propagation, the *source term* $g(q)$ is the radiance emitted by particles per unit of ray length, the *extinction function* $\tau(q)$ is the fraction of light attenuated by particles per unit of ray length, and $\frac{dC_\lambda(q)}{dq}$ is the instantaneous rate of change of $C_\lambda(q)$ with respect to q[18]. We recognize that the total change in radiance at distance q along the ray is the difference between the *source* term $g(q)$ controlling emission and the *attenuation* term $\tau(q)C_\lambda(q)$ controlling absorption.

With this physical interpretation, $q = 0$ is the far end of the ray at distance D from the observer, and $q = D$ is the position of the observer on the ray. Unfortunately, this ray metric is of limited value in the subsequent sections of this paper, and we would prefer to use a measuring system parameterized by a different variable, s, where $s = 0$ is the observer and $s = D$ is the far end of the ray. By reparameterizing

[1] To be exact, radiance is the flux transmitted per unit projected area per unit solid angle. It has units Φ/m^2sr, or watts per square meter per steradian. Curious readers and aspiring physicists are referred to chapter four of the free online optics textbook, *Physically Based Rendering: From Theory To Implementation*, by Pharr, Jakob, and Humphreys[22].

[2] In reality, another phenomenon called *scattering* exists, where light reflects between particles, sometimes adding radiance to our ray and sometimes attenuating it. For the sake of brevity, we bundle the light added through *in-scattering* into our concept of emission, and likewise, include *out-scattering* in our concept of attenuation. This simplifies our formulation.

every equation in terms of s, we can express radiance accumulation naturally as a function of distance from the observer. We use the identities $q = D - s$, and $\frac{dq}{ds} = -1$ to rewrite equation 2.1 in terms of s:

$$\begin{aligned}\frac{dC_\lambda(s)}{ds} &= \frac{dC_\lambda(q)}{dq}\frac{dq}{ds} \\ &= -\frac{dC_\lambda(q)}{dq} \\ &= \tau(q)C_\lambda(q) - g(q) \\ &= \tau(D-s)C_\lambda(D-s) - g(D-s).\end{aligned}$$

To simplify notation, we reuse the variables $C_\lambda(s)$, $g(s)$, and $\tau(s)$ to denote their corresponding values under the new distance parameterization. Now we have

$$\frac{dC_\lambda(s)}{ds} = \tau(s)C_\lambda(s) - g(s). \tag{2.2}$$

Equation 2.2 is solvable! Using the solution proved in section 4.1, we have

$$C_\lambda = \int_0^D c_\lambda(s)\tau(s)e^{-\int_0^s \tau(t)dt}ds. \tag{2.3}$$

where the source term is rewritten as $g(s) = c_\lambda(s)\tau(s)$ with the *emission coefficient* c_λ corresponding to the radiance at wavelength λ contributed per unit of extinction at distance s.

The *transmittance* term $e^{-\int_0^s \tau(t)dt}$ represents the attenuation incurred by the radiance contributed at distance s as it travels along the ray towards the observer. In other words, $g(s)$ adds some light to the viewing ray at distance s, and the fraction that completes its traversal of the ray is expressed by the transmittance term. By summing over the entire length of the ray with the outer integral, we effectively account for the total radiance contributed by each segment of the ray and the attenuation incurred by the radiance at each segment.

Because the bounds of the outer integral represent the distance from the observer, equation 2.3 integrates starting from the observer, moving outward to the arbitrary distance cutoff D. Hence, the direction of integration is exactly opposite to the direction of the light whose radiance we are calculating.

In practice, the functions τ and c_λ are often not defined by a closed-form expression, preventing us from computing an anti-derivative for equation 2.3. Instead, the integrals are computed numerically. The general idea is to split the outer integral into n smaller integrals[27] corresponding to sequential, nonoverlapping ray segments. Using this technique, we can approximate equation 2.3 with

$$C_\lambda \approx \sum_{i=1}^{n} c_\lambda(s_i)\left(1 - e^{-\tau(s_i)\delta_i}\right)T(i) \tag{2.4}$$

where

$$T(i) = \prod_{j=1}^{i-1} e^{-\tau(s_j)\delta_j}$$

represents the accumulated transmittance at the beginning of the ith segment. In equation 2.4, s_i represents the position on the ray of the start of the ith segment, and $\delta_i = s_{i+1} - s_i$ is the length of the ith segment. The proof of equation 2.4 is provided in section 4.2 of the appendix.

2.3 Volume Rendering

Using our findings from section 2.2, we will devise an outline for a renderer that aligns with equation 2.3. We will also show that such a renderer is inherently differentiable before discussing implementation strategies in section 2.4.

In volume rendering, a plane with its own coordinate system known as the *projection plane* is placed into the scene to act as an analog of the real computer screen on which the rendered image will be displayed. Similarly, a point known as the *center of projection* is placed behind the projection plane to represent the position of a human eye observing the scene. The perpendicular distance (or the shortest distance) between the projection plane and the center of projection is known as the *focal length*[3].

[3]Short focal lengths result in a large field of view, distorting the scene with a wide-angle lens effect, whereas large focal lengths narrow the field of view and produce a magnifying effect.

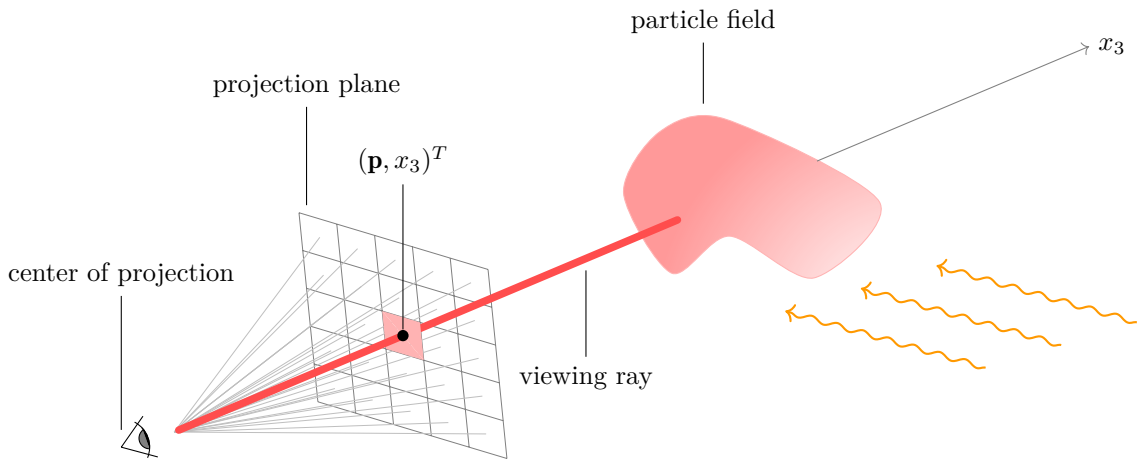

Figure 2.2: **Volume rendering**. A reddish-pink particle field is projected to a single pixel $\mathbf{p}$ on the projection plane. The light source is off screen as indicated by the wavy arrows. All coordinates and axis labels are given in ray space, but the diagram is shown in camera space for clarity.

Every pixel on the computer screen corresponds to a unique area on the projection plane surrounding a coordinate $\mathbf{p}$. Our goal, therefore, is to calculate the radiance along every *viewing ray* that passes through the scene, intersects $\mathbf{p}$, and converges at the center of projection (see figure 2.2). Importantly, the viewing ray intersecting $\mathbf{p}$ exists, is unique, and directly corresponds to the pixel value we seek to calculate[31]. Volume renderers calculate the value of every pixel on the screen by computing the radiance at the center of projection of every viewing ray[4].

First, we must convert the volume data for the scene into a format that is compatible with this approach. We assume that the scene is initially represented with *world space* coordinates, where the origin has no semantic meaning (perhaps it is at the "center" of the scene). The first step is to apply the affine transformation φ which maps the volume data to a coordinate system known as *camera space*. In camera space, the origin corresponds to the center of projection, the negative z-direction represents the viewing angle (or the "forward" direction from the observer's perspective), and the positive y-direction points "up" from the observer's perspective.

Next, we apply the non-affine transformation ϕ which maps the scene to a non-Cartesian coordinate system known as *ray space*. In ray space, viewing rays are parallel to a coordinate axis known as the *depth axis* (see figure 2.3). Ray space coordinates are extremely well behaved: A point $\mathbf{x} = (x_1, x_2, x_3)^T$ lies x_3 units of distance from the center of projection and is necessarily intersected by exactly one viewing ray, r, which also intersects the projection plane at $(x_1, x_2)^T$. We can also denote the same ray space coordinate with $\mathbf{x} = (\mathbf{p}, x_3)^T$, where $\mathbf{p}$ is the coordinate on the projection plane intersected by r, and x_3 is the signed Euclidean distance of $\mathbf{x}$ from the center of projection[31].

By converting the scene data to ray space, the viewable portion of the scene can be projected to the projection plane, after which the projected elements of the scene reside in *screen space*, the coordinate system of the projection plane. In some sources, the units of screen space refer to a pixel coordinate within the domain $[0, W] \times [0, H]$ where W and H are the number of pixels in the width and height of the projection plane respectively. In other sources, the units of screen space may refer to *normalized device coordinates* (NDC) within the domain $[-1, 1] \times [-1, 1]$. In the context of this thesis, the units applied to screen space will switch depending on context and will be specified when a distinction affects meaning.

To project the scene from ray space to screen space, we may integrate along the depth axis and apply equation 2.3 to calculate the radiance at a given wavelength along a pixel's viewing ray. This application of equation 2.3 effectively transforms the scene's volume data to screen space and completes the projection pipeline. If we assume that the appearance of the scene at wavelength λ can be completely represented by

[4]It is assumed that none of the scene's medium is present between the center of projection and the projection plane, and thus no radiance is emitted or attenuated along any viewing ray between the projection plane and the center of projection.

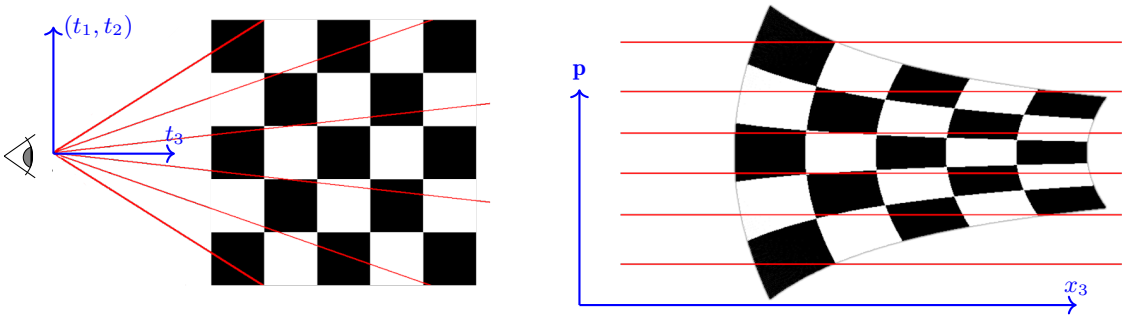

Figure 2.3: **Ray space.** Left: Camera space coordinate scheme. Right: The same image but in ray space after applying the 2D analog of ϕ. The vertical axis represents movement along the projection plane, and horizontal movement represents distance from the center of projection[31]. To generate the figure, the ϕ operator was implemented using Numpy[6]. This figure is a recreation of a figure in Zwicker et al.[31]

an extinction function $f'_c(\mathbf{p}, x_3) = f(\phi(\varphi(\mathbf{x})))$ and emission coefficient $c_\lambda(\mathbf{p}, x_3)$, then we may reformulate the emission-absorption equation (equation 2.3) in ray space to yield the volume rendering equation:

$$C_\lambda(\mathbf{p}) = \int_0^D c_\lambda(\mathbf{p}, \xi) f'_c(\mathbf{p}, \xi) e^{-\int_0^\xi f'_c(\mathbf{p}, \rho) d\rho} d\xi. \tag{2.5}$$

Similarly, we can redefine the emission-absorption approximation (equation 2.4) in terms of $f'_c(\mathbf{p}, x_3)$ and $c_\lambda(\mathbf{p}, x_3)$ with:

$$C_\lambda(\mathbf{p}) \approx \sum_{i=1}^{n} c_\lambda(\mathbf{p}, s_i) \left(1 - e^{-f'_c(\mathbf{p}, s_i)\delta_i}\right) T(\mathbf{p}, i) \tag{2.6}$$

where

$$T(\mathbf{p}, i) = \prod_{j=1}^{i-1} e^{-f'_c(\mathbf{p}, s_j)\delta_j}.$$

Crucially, a renderer using equation 2.6 is fully differentiable! Given a scene parameter θ_i, our eventual goal will be to calculate $\frac{\partial \mathcal{L}}{\partial \theta_i}$ where $\mathcal{L}(\mathbf{I}, \widehat{\mathbf{I}})$ is a differentiable loss function, $\mathbf{I}$ is a matrix of the calculated radiance for every pixel on the projection plane, and $\widehat{\mathbf{I}}$ is a ground truth image. The gradient $\frac{\partial \mathcal{L}}{\partial \theta_i}$ can be calculated as the sum of the pixel-wise gradients, written

$$\frac{\partial \mathcal{L}}{\partial \theta_i} = \sum_{\mathbf{p} \in P} \frac{\partial \mathcal{L}}{\partial C_\lambda(\mathbf{p})} \frac{\partial C_\lambda(\mathbf{p})}{\partial \theta_i}.$$

When calculating $C_\lambda(\mathbf{p})$ with equation 2.6, we can split $\frac{\partial C_\lambda(\mathbf{p})}{\partial \theta_i}$ into the sum of gradients from each sample point along the viewing ray of $\mathbf{p}$ with

$$\frac{\partial C_\lambda(\mathbf{p})}{\partial \theta_i} = \sum_{k=1}^{n} \left(\frac{\partial C_\lambda(\mathbf{p})}{\partial f'_c(\mathbf{p}, s_k)} \frac{\partial f'_c(\mathbf{p}, s_k)}{\partial \theta_i} + \frac{\partial C_\lambda(\mathbf{p})}{\partial c_\lambda(\mathbf{p}, s_k)} \frac{\partial c_\lambda(\mathbf{p}, s_k)}{\partial \theta_i} \right).$$

Combining results, we have

$$\frac{\partial \mathcal{L}}{\partial \theta_i} = \sum_{\mathbf{p} \in P} \frac{\partial \mathcal{L}}{\partial C_\lambda(\mathbf{p})} \sum_{k=1}^{n} \left(\frac{\partial C_\lambda(\mathbf{p})}{\partial f'_c(\mathbf{p}, s_k)} \frac{\partial f'_c(\mathbf{p}, s_k)}{\partial \theta_i} + \frac{\partial C_\lambda(\mathbf{p})}{\partial c_\lambda(\mathbf{p}, s_k)} \frac{\partial c_\lambda(\mathbf{p}, s_k)}{\partial \theta_i} \right).$$

Because equations 2.5 and 2.6 are differentiable with respect to f'_c and c_λ, we can easily calculate $\frac{\partial C_\lambda(\mathbf{p})}{\partial f'_c(\mathbf{p}, s_k)}$ and $\frac{\partial C_\lambda(\mathbf{p})}{\partial c_\lambda(\mathbf{p}, s_k)}$. Furthermore, we already know that $\frac{\partial \mathcal{L}}{\partial C_\lambda}$ is available because $\mathcal{L}$ is differentiable. As long as f'_c and c_λ are defined to be differentiable with respect to an arbitrary scene parameter θ_i, any volume renderer that implements equation 2.6 will be fully differentiable.

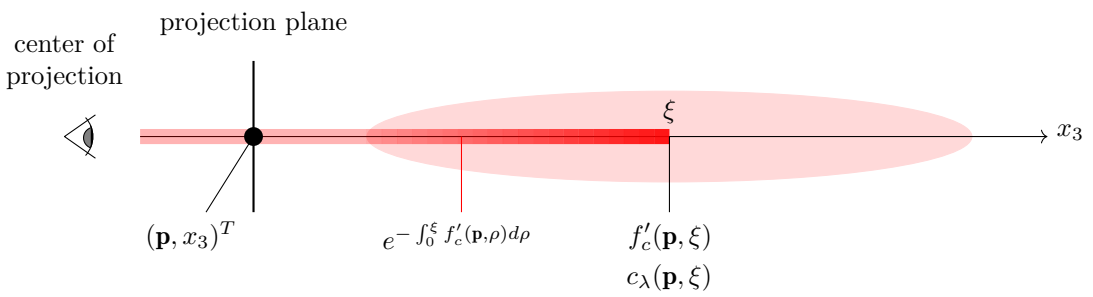

Figure 2.4: **Transmittance**. The side view of the highlighted ray and particle field from figure 2.2 is depicted with the transmittance and source terms labeled. The effect of the transmittance term on the radiance emitted at position ξ on the view ray is demonstrated. The particle field attenuates much of the radiance contributed by the ray segment at ξ.

2.4 Representations

The choice of parameterization used to define the scene parameters, commonly referred to as the data *representation*, introduces limitations in differentiable rendering. Of the many approaches that have been tried, four distinct categories of data representation emerge: meshes, neural implicit functions, voxels, and point clouds. We will examine each approach briefly, presenting each as a solution to the shortcomings of the last, and weighing it against our results from sections 2.2 and 2.3. We will find that meshes are inherently non-differentiable, neural implicit representations are differentiable but prohibitively slow, voxels are fast but memory-inefficient, and point clouds offer the most promising foundation.

2.4.1 Mesh

Traditional rendering pipelines often represent the scene using a set of vertices connected by a separate set of facets that form a triangle *mesh*. Meshes have historically been popular because they can store complex geometry compactly and render quickly on modern graphics hardware.

Unlike solid-based volume rendering, meshes are a surface-based representation, meaning that only the surfaces of objects are represented. Instead of utilizing volume rendering, therefore, mesh-based renderers use an entirely different approach. First, every triangle is projected from world space to screen space, and each pixel $\mathbf{p}$ is assigned the closest triangle to the screen that contains it. Next, the color of $\mathbf{p}$ is interpolated using the colors attributed to each vertex of its assigned triangle[13].

Although a mesh representation may seem like a natural choice for differentiable rendering because of its popularity in other real-time rendering applications[1, p. 699], meshes are inherently non-differentiable. The issue is that the triangle selection process is discrete in nature, whereas only continuous functions can be differentiable[13]. Furthermore, *background* pixels that are not contained within any projected facet of the mesh cannot contribute to the gradient of the vertices or their attributes. Other *foreground* pixels are only able to contribute to the gradients of the vertices which define their bounding facets. Several works[3, 12, 17] address these issues with varying degrees of success, but the existence of an ideal solution to the various discontinuities imposed by mesh rendering remains uncertain[13].

In the context of this paper, mesh representations expose the challenges of designing differentiable renderers that do not conform to the volume rendering template proposed in previous sections.

2.4.2 Neural Implicit

To implement volume rendering, the scene must be represented by $f'_c : \mathbb{R}^2 \times \mathbb{R} \to \mathbb{R}$ and $c_\lambda : \mathbb{R}^2 \times \mathbb{R} \to \mathbb{R}$. Neural implicit rendering defines f'_c and c_λ through a neural network F_Θ that takes as input a position in world space $\mathbf{x} = (x_1, x_2, x_3)^T$ and maybe a Cartesian vector $\mathbf{v}$ describing the viewing angle. As output, F_Θ provides the emission coefficient $c_\lambda(\mathbf{p}, x_3)$ and extinction coefficient $f'_c(\mathbf{p}, x_3)$. The weights defining the network are the scene parameters that the renderer will train until the scene converges.

Position alone is often insufficient to determine emission coefficients in a scene because it does not account for the viewing direction. By providing F_Θ with the vector $\mathbf{v}$ describing the viewing ray, view-

dependent specular effects can be considered when training the network. In practice, completely learning the reflection characteristics of glossy surfaces is often intractable, but adding view-dependent highlights to surfaces can hint at the direction of the light source and make the scene seem more realistic. For example, F_Θ can learn that the crests of ocean waves tend to glint when viewed from certain angles.

In large part, the appeal of neural implicit rendering comes from how directly it follows from volume rendering theory. The neural network F_Θ is a truly continuous function that immediately facilitates the use of equation 2.6. In addition, neural networks are well studied, and sophisticated techniques for training them have received much attention in recent years.

Neural implicit rendering is effective too! For a few years after it was published in 2020, the neural implicit rendering implementation specified in Neural Radiance Fields (NeRF) by Mildenhall et al.[19] was used as a state-of-the-art benchmark[29, 14]. In NeRF, they train two identical networks simultaneously: a "coarse" network and a "fine" network[5]. The idea was to limit the number of samples required per viewing ray by sampling the coarse network uniformly at random. This coarse sampling would serve to predict the location of the densest areas of the scene passed through by the ray. Detail could then be inferred through an importance-weighted sampling of the fine network along the same ray.

The main problem with NeRF was its lack of speed during both training and rendering. To compute equation 2.6 for a viewing ray, the coarse and fine networks would be queried 64 and 192 times respectively for a total of 256 queries per ray. For their dataset, Mildenhall et al. used 762k rays per image totalling around 200 million queries per rendered image, or about 30 seconds of processing time on an NVIDIA V100. Worse still, a single scene took 100-300k iterations to train using a batch size of 4096 rays, or about 1-2 days on an NVIDIA V100[19].

2.4.3 Voxel

A voxel is a unit cube in world space that is functionally analogous to a 3D pixel. In principle, the entire scene can be represented by dividing world space into voxels, each containing some piece of information about the finite portion of the scene it occupies. With voxel representations, points along a ray can be sampled by reading values from each intersected voxel, or by interpolating between nearby voxels. Because voxels are easily stored in memory as a conventional data structure, retrieving samples along a ray is substantially faster than with neural implicit approaches like NeRF. As such, voxel representations are well-suited for volume rendering and offer a promising alternative to slower neural implicit approaches.

In 2021, about a year after NeRF, Yu et al. published Plenoxels[29]—a voxel-based approach that came remarkably close to matching NeRF's performance while reducing training/optimization time by roughly 100×. To represent the scene, Yu et al. used a dense array in which each element corresponded to the volume occupied by a given voxel. As such, the elements of the dense array were a mixture of pointers and null values to represent occupied and unoccupied portions of the scene respectively. The pointers in the dense array pointed to an array of voxel values, each containing an opacity and a vector of *spherical harmonic* coefficients.

Plenoxels uses spherical harmonics to capture the same specular effects as NeRF. Spherical harmonics are a set of orthogonal functions on the sphere that act as a basis; any function on the sphere can be expressed as a weighted sum of them. In a scene representation that uses spherical harmonics, color is not expressed as an RGB vector, but rather as a vector of spherical harmonic coefficients that parameterize color as a function of viewing direction. Spherical harmonics are particularly useful in the context of differentiable rendering because they are easy to differentiate with respect to their optimizable coefficients.

Using dense arrays to represent voxels leads to a fundamental issue where each finite portion of the scene must be represented in memory regardless of whether or not it is occupied. As such, only small scenes can be represented with a high degree of detail. To further illustrate the wastefulness of voxels, consider how the number of voxels required to represent a scene scales with $O(n^3)$ while the area of the surfaces in the scene we seek to represent only scales with $O(n^2)$[20]. Also, because all voxels are unit sized, resolution is distributed uniformly across the scene regardless of local detail. An ideal approach would scale to represent fine or coarse textural detail within scenes, but voxels do not offer a clear mechanism for this behavior[20].

[5]Both coarse and fine networks use the same architecture with 9 fully connected hidden layers, each with 256 channels, and a final 10th fully connected layer with 128 channels.

2.4.4 Point Cloud

With point cloud representations, the scene is represented as a set of 3D point coordinates in world space. Typically, point clouds are rendered by projecting the points to screen space and calculating the influence of relevant projected points on the color of each pixel[13]. This approach aligns well with the principles of volume rendering because it is essentially an abstraction of the same concept. In volume rendering, the casting of a viewing ray through a pixel is analogous to calculating the influence of each point in a point cloud on the color of that pixel. In fact, section 2.5 will discuss how one particularly powerful point cloud approach can be derived directly from volume rendering.

Point clouds address many of the issues present in other representations. Because the point projection step is performed using simple arithmetic, the rendering pipeline is easily differentiable, unlike mesh-based approaches. Also, point clouds are simple data structures that do not depend on expensive neural network inferences like those used in NeRF-like implicit approaches. Finally, unlike voxels, points in a point cloud can be placed arbitrarily in space, allowing for memory efficiency and flexibility in representing fine textural details.

2.5 3D Gaussian Splatting

3D Gaussian splatting (3DGS) is a point-cloud-based approach where each point is the mean of a 3D Gaussian function in world space augmented with a color and an opacity value. During rendering, each 3D Gaussian in world space is projected to form a 2D Gaussian in screen space. Ultimately, the color of each pixel is calculated as a weighted sum of the Gaussians' colors, where the weights are determined by evaluating the projected Gaussians at the pixel's position in screen space.

Put plainly, a 3D Gaussian augmented with color and opacity values appears visually similar to a translucent ellipse, opaque at its center and becoming increasingly transparent in world space (see figure 2.5). By composing the scene using hundreds of thousands or millions of these colored ellipses, we can achieve a very fine degree of detail while also easily calculating the color of each pixel based on how the Gaussians overlap from a given camera pose. If one were to compare a 3DGS scene to an oil painting, then each Gaussian would be the three-dimensional equivalent of a single brush stroke with a specific color, size, rotation, and position. By iteratively optimizing these parameters of each Gaussian, the scene is gradually "painted" (recall figure 2.1).

In this section, we derive an algorithm for calculating the color of a pixel given a set of projected Gaussians. We then develop a separate algorithm for projecting world space Gaussians to screen space, which allows the first algorithm to be applied.

3DGS was first described in detail by Kerbl et al.[14], building off prior work by Zwicker et al.[31]. The approach showcased in this section will track closely with both papers.

2.5.1 Calculating Pixel Color

First, we define a 3D Gaussian function in world space with

$$\mathcal{G}^3_\Sigma(\mathbf{x}-\boldsymbol{\mu}) = \frac{1}{(2\pi)^{3/2}|\Sigma|^{1/2}} e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu})} \tag{2.7}$$

where Σ is the 3×3 covariance matrix describing the Gaussian's shape in world space, $|\Sigma|$ is the determinant of Σ, and $\boldsymbol{\mu} = (\mu_1, \mu_2, \mu_3)^T$ is the Gaussian's mean in world space. When defined in this form, we have $1 = \int_{\mathbb{R}^3} \mathcal{G}^3_\Sigma(\mathbf{x}-\boldsymbol{\mu})d\mathbf{x}$. For a depiction of a 3D Gaussian, refer to figure 2.5.

Later in this section, we will also define 3D Gaussian functions in ray space. In those cases, we will define ray space Gaussians using the same formula as for world space Gaussians but using $\boldsymbol{\mu}'$ and Σ' (instead of $\boldsymbol{\mu}$ and Σ) to distinguish them from their world space counterparts. Similarly, we will handle screen space Gaussian functions which have a different definition entirely:

$$\mathcal{G}^2_{\hat{\Sigma}}(\mathbf{p}-\hat{\boldsymbol{\mu}}) = \frac{1}{2\pi|\hat{\Sigma}|^{1/2}} e^{-\frac{1}{2}(\mathbf{p}-\hat{\boldsymbol{\mu}})^T\hat{\Sigma}^{-1}(\mathbf{p}-\hat{\boldsymbol{\mu}})} \tag{2.8}$$

where $\mathbf{p}$ is the screen space coordinate of a pixel, $\hat{\Sigma}$ is a 2×2 covariance matrix, and $\hat{\boldsymbol{\mu}} = (\hat{\mu}_1, \hat{\mu}_2)^T$ is the screen space mean of the Gaussian. With the 2D definition, we also have $1 = \int_{\mathbb{R}^2} \mathcal{G}^2_{\hat{\Sigma}}(\mathbf{p}-\hat{\boldsymbol{\mu}})d\mathbf{p}$[31].

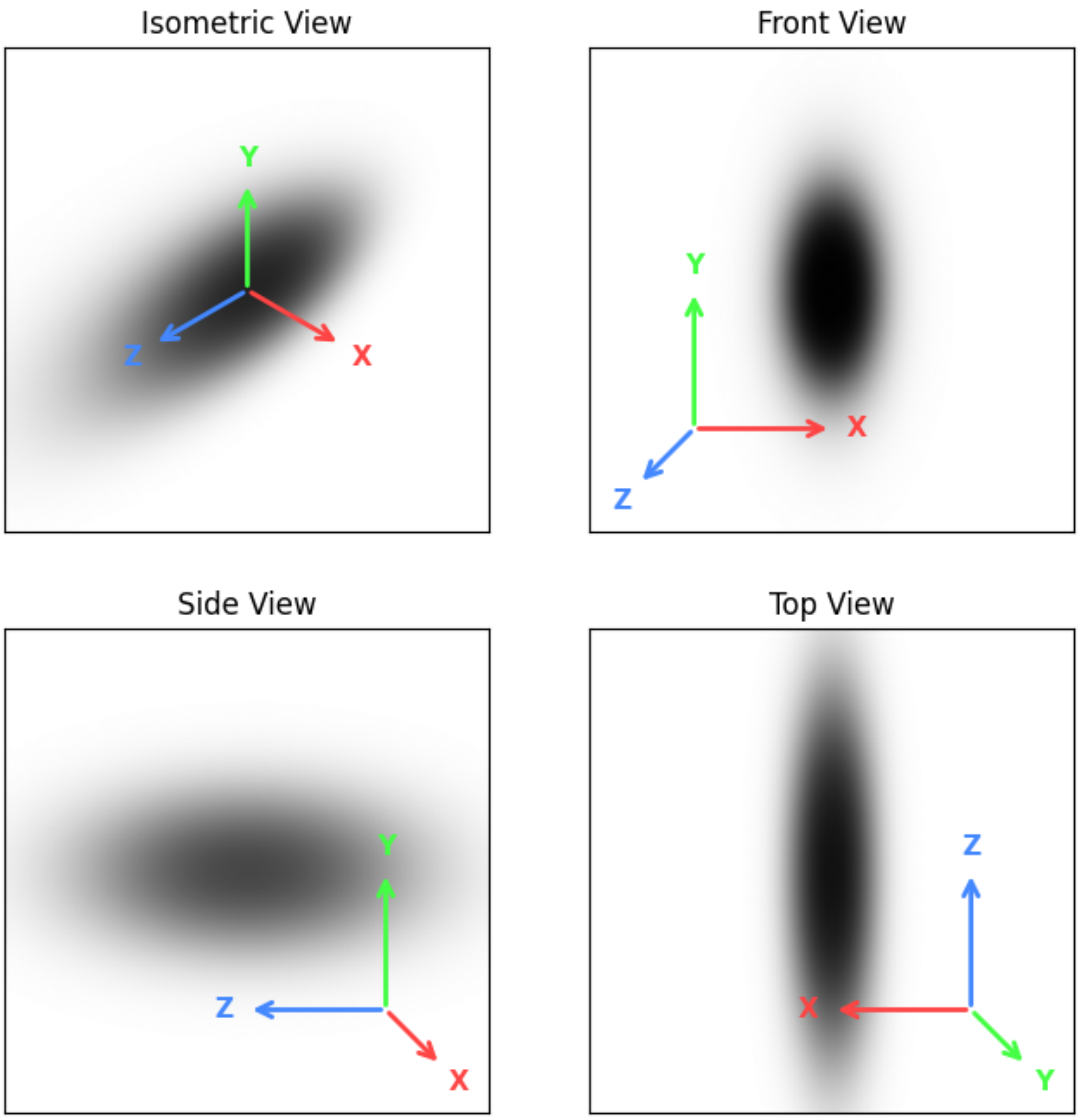

Figure 2.5: **Volume Rendering of a Gaussian** with $\boldsymbol{\mu} = (0, 0, 0)^T$ and $\Sigma = \begin{bmatrix} 0.0144 & 0 & 0 \\ 0 & 0.625 & 0 \\ 0 & 0 & 0.25 \end{bmatrix}$ corresponding to a scaling factor $s = (0.12, 0.25, 0.5)^T$ where a standard deviation is 0.12 units in the world space x direction, 0.25 units in y direction, and 0.5 units in the z direction. The axes meet at $p = (-1, -1, -1)^T$ in world space and are each 1 unit of length in world space. In each frame, the camera is 2.5 units of Euclidean distance (in world space) from the origin with a focal length of 2. The volume renderer used to generate this rendering is available at https://github.com/UncatchableAlex/Thesis_Diagrams/tree/main/gaussian_lla.

We now wish to derive the splatting formula for calculating pixel color given a set of projected Gaussians. Recall equation 2.6

$$C_\lambda(\mathbf{p}) \approx \sum_{i=1}^{n} c_\lambda(\mathbf{p}, s_i) \left(1 - e^{-f_c'(\mathbf{p},s_i)\delta_i}\right) \left(\prod_{j=1}^{i-1} e^{-f_c'(\mathbf{p},s_j)\delta_j}\right)$$

which approximates the volume rendering formula by splitting the viewing ray for each $\mathbf{p}$ into n discrete segments. With splatting, instead of splitting the viewing ray into arbitrary segments, we wish to assign one segment per Gaussian in the scene.

For this substitution to work, we have to make three critical assumptions. First, we assume that each Gaussian has *local support*, meaning that even though a Gaussian $\mathcal{G}^3_\Sigma(\mathbf{x} - \boldsymbol{\mu}) > 0$ for all $\mathbf{x}$, we will approximate its value as 0 for all $\mathbf{x}$ further than some distance from $\boldsymbol{\mu}$. Second, we assume that the local support regions of two Gaussians never overlap[6], and the Gaussians are ordered front to back by depth. Lastly, we assume that the emission coefficient of a Gaussian is constant throughout its support. We define the emission coefficient of the ith Gaussian at the wavelength λ as $c_{i,\lambda}$[31, 14].

In equation 2.6, the expression $f_c'(\mathbf{p}, s_i)\delta_i$ treats f_c' as a constant over each segment spanning $\delta_i = s_{i+1} - s_i$ in length. More formally, equation 2.6 uses $f_c'(\mathbf{p}, s_i)\delta_i$ as an approximation for $\int_{s_i}^{s_{i+1}} f_c'(\mathbf{p}, \xi)d\xi$ because

$$\int_{s_i}^{s_{i+1}} f_c'(\mathbf{p}, \xi)d\xi \approx \int_{s_i}^{s_{i+1}} f_c'(\mathbf{p}, s_i)d\xi = f_c'(\mathbf{p}, s_i)\delta_i.$$

In 3DGS, when we replace the ray segment s_i with the ith Gaussian $g_i(\mathbf{x}) = \mathcal{G}^3_{\Sigma_i}(\mathbf{x} - \boldsymbol{\mu}_i)$, we define

$$f_c'(\mathbf{p}, \xi) = \sigma_i g_i'(\mathbf{p}, \xi)$$

where σ_i is the opacity value associated with g_i, and $g_i'(\mathbf{p}, x_3) = g_i(\varphi^{-1}\phi^{-1}(\mathbf{p}, x_3))$ is g_i projected to ray space. Also, we can let the integral bounds span all positive numbers because g_i' has local support. These modifications lead us directly to the equation

$$\int_{s_i}^{s_{i+1}} f_c'(\mathbf{p}, \xi)d\xi = \sigma_i \int_0^\infty g_i'(\mathbf{p}, \xi)d\xi \tag{2.9}$$

Substituting equation 2.9 into equation 2.6, we have

$$C_\lambda(\mathbf{p}) \approx \sum_{i=1}^{n} c_{i,\lambda} \left(1 - e^{-\sigma_i \hat{g}_i(\mathbf{p})}\right) \left(\prod_{j=1}^{i-1} e^{-\sigma_j \hat{g}_j(\mathbf{p})}\right) \tag{2.10}$$

where

$$\hat{g}_i(\mathbf{p}) = \int_0^\infty g_i'(\mathbf{p}, \xi)d\xi \tag{2.11}$$

is a Gaussian function in screen space with mean $\hat{\boldsymbol{\mu}}_i$ and covariance $\hat{\Sigma}_i$. The last step is to approximate the exponentials in 2.10 using the first two terms of the Maclaurin series $e^{-\sigma_i \hat{g}_i(\mathbf{p})} \approx 1 - \sigma_i \hat{g}_i$. Using this substitution, we are left with the spectral pixel color formula for 3DGS:

$$C_\lambda(\mathbf{p}) \approx \sum_{i=1}^{n} c_{i,\lambda}\sigma_i \hat{g}_i(\mathbf{p}) \left(\prod_{j=1}^{i-1} (1 - \sigma_j \hat{g}_j(\mathbf{p}))\right). \tag{2.12}$$

Unfortunately, scenes with color emit a full spectrum of light visible along every viewing ray. Calculating these spectra from point-sampled radiance computations using equation 2.12 is inefficient and not aligned with the photographic techniques[11] used to capture the ground truth images used by 3DGS for training. As such, we modify equation 2.12 to calculate a *tristimulus value* along a viewing ray for a given *sensor sensitivity* function k as described in section 4.3. Furthermore, 3DGS uses a vector of spherical harmonic coefficients to encode each tristimulus value for each Gaussian. As in Plenoxels[29],

[6]The reason why the Gaussians must not overlap is that the transmittance calculation complicates if they do. Say the kth Gaussian is transmitting radiance along $\mathbf{p}$'s viewing ray, but it is overlapped by the jth Gaussian. It then becomes difficult to calculate how much of the radiance from the kth Gaussian is attenuated by the jth Gaussian. It is much simpler to say that all of the radiance contributed by the kth Gaussian interacts with all of the jth Gaussian, and determine from there how much radiance is transmitted.

the use of spherical harmonics allows Gaussians to express color as a function of the viewing direction $\mathbf{v}$, enabling them to model specular effects. We ultimately end up with the tristimulus pixel color formula with spherical harmonics:

$$C_k(\mathbf{p}, \mathbf{v}) \approx \sum_{i=1}^{n} c_{i,k}(\mathbf{v}) \sigma_i \hat{g}_i(\mathbf{p}) \left(\prod_{j=1}^{i-1} (1 - \sigma_j \hat{g}_j(\mathbf{p})) \right), \quad k \in \{r, g, b\}. \tag{2.13}$$

Before trying to write an algorithm implementing equation 2.13, the following observations should be made:

1. In order to calculate equation 2.13, we need the emission functions $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_k$ where $\mathbf{c}_i = (c_{i,r}, c_{i,g}, c_{i,b})^T$. We also need the opacity values $\sigma_1, \ldots, \sigma_n$ that augment the Gaussians. To define the Gaussian functions themselves, we require the screen space means $\hat{\boldsymbol{\mu}}_1, \ldots, \hat{\boldsymbol{\mu}}_n$, and the screen space covariant matrix inverses, also known as the *conics* $\hat{\Sigma}_1^{-1}, \ldots, \hat{\Sigma}_n^{-1}$.

2. The transmittance term of equation 2.12, $T_i = \prod_{j=1}^{i-1}(1 - \sigma_j \hat{g}_j(\mathbf{p}))$, can be tracked through the recursive relation $T_{i+1} = T_i(1 - \sigma_j \hat{g}_j(\mathbf{p}))$.

3. There is no need to calculate the normalization coefficient of $\hat{g}_i(\mathbf{p})$ given in definition 2.8. Since this coefficient is constant with respect to $\mathbf{p}$, it can be absorbed into the optimizable opacity parameter. Defining
$$\tilde{\sigma}_i = \frac{\sigma_i}{2\pi |\hat{\Sigma}_i|^{1/2}},$$
we may write
$$\sigma_i \hat{g}_i(\mathbf{p}) = \tilde{\sigma}_i e^{(\mathbf{p} - \hat{\boldsymbol{\mu}}_i)\hat{\Sigma}^{-1}(\mathbf{p} - \hat{\boldsymbol{\mu}}_i)^T} \tag{2.14}$$
and, for simplicity, we continue to denote the reparameterized operator $\tilde{\sigma}_i$ as σ_i.

4. It may be unnecessary to sum the radiance contributions of all n Gaussians. Once $T_i < \epsilon$ for some small $\epsilon > 0$, the entire expression inside the summation will be so small that it can be ignored, allowing the algorithm to terminate early.

5. The equation 2.13 can be applied to all k values simultaneously for scenes with color. We should assume that each Gaussian has constant (with respect to the viewing angle) emission coefficients defined for r, g, and b. To facilitate the calculation of multiple tristimulus values simultaneously, the variables $\mathbf{c}_i$, C, and T should be defined as vectors of size 3 in the pseudocode.

6. If there are n Gaussians in the scene, it should be possible to only use a subset of them in calculating equation 2.13 for a given pixel $\mathbf{p}$. Many Gaussians will evaluate to 0 at $\mathbf{p}$, so if the Gaussians are cleverly sorted, then $\hat{g}_i(\mathbf{p}) > 0$ only for $i_{\text{start}} \leq i < i_{\text{end}}$.

7. If every Gaussian with indices $i_{\text{start}} \leq i < i_{\text{end}}$ has been considered and the final transmittance $T_{\text{final}} > \epsilon$, then the remaining radiance along the viewing ray for $\mathbf{p}$ should transmit the color of the scene's background. We will call this background color C_{bg}.

Using these observations, we can now construct algorithm 1, the Forward Pixel Color Algorithm for 3DGS. See algorithm 1.

2.5.2 Projecting Gaussians From World Space to Ray Space

Equation 2.13 requires the ith Gaussian projected to ray space, which by definition is given by

$$\begin{aligned} g_i'(\mathbf{r}) &= g_i(\varphi^{-1}(\phi^{-1}(\mathbf{r}))) \\ &= \mathcal{G}^3_{\Sigma_i}(\varphi^{-1}(\phi^{-1}(\mathbf{r})) - \boldsymbol{\mu}_i), \end{aligned} \tag{2.15}$$

where $\mathbf{r}$ is a ray space coordinate. However, we have yet to establish a closed-form expression for g_i' that defines its covariance matrix Σ_i' and mean $\boldsymbol{\mu}_i'$ in terms of Σ_i, and $\boldsymbol{\mu}_i$. Recall that φ is the transformation from world space to camera space and ϕ is the transformation from camera space to ray space. The composition of ϕ and φ results in a transformation from world space to ray space, denoted with $\mathbf{r} = (\phi \circ \varphi)(\mathbf{x}) = \phi(\varphi(\mathbf{x}))$. Conversely, $\mathbf{x} = (\phi \circ \varphi)^{-1}(\mathbf{r}) = \varphi^{-1}(\phi^{-1}(\mathbf{r}))$ is the transformation from ray space to world space.

Algorithm 1 Forward Pixel Color (Implementing Equation 2.13)[14]

Require: $n > 0$, $\epsilon > 0$
Opacities: $\sigma_1, \ldots, \sigma_n$
Conics: $\hat{\Sigma}_1^{-1}, \ldots, \hat{\Sigma}_n^{-1}$
Colors: $\mathbf{c}_1, \ldots, \mathbf{c}_n$ where $\mathbf{c}_i = (c_{i,r}, c_{i,g}, c_{i,b})^T$
Screen-space means: $\hat{\boldsymbol{\mu}}_1, \ldots, \hat{\boldsymbol{\mu}}_n$
Pixel position: $\mathbf{p}$
Indices: $i_{\text{start}}, i_{\text{end}}$ where $0 < i_{\text{start}} \leq i_{\text{end}}$
Background color: C_{bg}
Viewing direction: $\mathbf{v}$

$T_{i_{\text{start}}} \leftarrow 1$ ▷ Initialize the transmittance as 1
$C \leftarrow (0, 0, 0)^T$ ▷ Initialize the final color with 0 for each tristimulus value
$i \leftarrow i_{\text{start}}$
while $T_i > \epsilon$ and $i < i_{\text{end}}$ **do**
  $\mathbf{d}_i \leftarrow \mathbf{p} - \hat{\boldsymbol{\mu}}_i$ ▷ The distance from $\mathbf{p}$ to $\hat{\boldsymbol{\mu}}_i$
  $\alpha_i \leftarrow \sigma_i \exp(-\frac{1}{2}\mathbf{d}_i^T \hat{\Sigma}_i^{-1} \mathbf{d}_i)$ ▷ equation 2.14
  for $k \in [r, g, b]$ **do**
    $C_k \leftarrow C_k + \alpha_i c_{i,k}(\mathbf{v}) T_i$ ▷ Update the net emission of tristimulus value k
  end for
  $T_{i+1} \leftarrow T_i(1 - \alpha_i)$ ▷ Update the transmittance
  $i \leftarrow i + 1$
end while
$T_{final} \leftarrow T_i$
$C_{\text{final}} \leftarrow C + T_{\text{final}} C_{\text{bg}}$ ▷ Whatever transmittance remains will transmit the background color
$i_{\text{final}} \leftarrow i - 1$ ▷ The index of the last Gaussian touched
return C_{final}, T_{final}, i_{final}

One of the reasons 3DGS uses Gaussian functions is because Gaussians are *closed* under arbitrary affine transformations. In other words, applying the affine transformation Φ to a Gaussian results in another Gaussian! Because Φ is affine, we can define it as $\Phi(\mathbf{x}) = \mathbf{M}\mathbf{x} + \mathbf{c}$ where $\mathbf{M}$ is an $n \times n$ matrix and $\mathbf{c} = (c_1, \ldots, c_n)^T$ is a column vector. Using Φ, the Gaussian function affine transformation formula[31] is given as

$$\mathcal{G}_{\mathbf{V}}^n(\Phi^{-1}(\mathbf{u}) - \boldsymbol{\mu}) = \frac{1}{|\mathbf{M}^{-1}|}\mathcal{G}_{\mathbf{MVM}^T}^n(\mathbf{u} - \Phi(\boldsymbol{\mu})). \tag{2.16}$$

We would now like to define $\Phi = \phi \circ \varphi$ as an affine transformation, but because ϕ is non-affine, Φ is non-affine as well[7]. However, we can define the viewing transformation φ with the affine transformation $\varphi(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b}$, where $\mathbf{W}$ is a rotation matrix and $\mathbf{b}$ is a translation vector, but unfortunately there is no similar construction for ϕ. Instead, we define ϕ and its inverse ϕ^{-1} with

$$\begin{aligned} \mathbf{r} = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \end{pmatrix} = \phi(\mathbf{t}) \quad &= \begin{pmatrix} f_x \cdot t_1/t_3 \\ f_y \cdot t_2/t_3 \\ ||(t_1, t_2, t_3)^T|| \end{pmatrix} \\ \mathbf{t} = \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix} = \phi^{-1}(\mathbf{r}) &= \begin{pmatrix} (r_3 \cdot r_1)/(f_x \cdot \ell) \\ (r_3 \cdot r_2)/(f_y \cdot \ell) \\ r_3/\ell \end{pmatrix} \end{aligned} \tag{2.17}$$

where $\ell = ||(r_1/f_x, r_2/f_y, 1)^T||$, $\mathbf{x}$ is a world space coordinate, $\mathbf{t} = \varphi(\mathbf{x})$ is a camera space coordinate, and $\mathbf{r} = \phi(\mathbf{t})$ is a ray space coordinate [31, 14]. The variables f_x and f_y are the focal length f expressed in pixel width and pixel height respectively. Both f_x and f_y are needed because the rectangular pixels that divide the projection plane may have different sizes in their x and y directions due to various imperfections in the camera used to capture the ground truth images[25]. The equation for $\phi(\mathbf{t})$ works due to similar triangles. For example, r_1 uses the proportion $\frac{r_1}{f_x} = \frac{t_1}{t_3}$[8]. Ultimately, r_1 and r_2 are measured in pixel units from the center of the projection plane.

[7] To prove this claim more rigorously, we proceed by contraposition: Suppose that $\Phi = \phi \circ \varphi$ and φ are affine but ϕ is non-affine, then $\Phi \circ \varphi^{-1} = \phi$ would also be affine because affine functions are closed under composition and inversion. However, we are given that $\phi = \Phi \circ \varphi^{-1}$ is non-affine, so Φ must be non-affine too.

[8] Curious readers are encouraged to draw a diagram detailing the center of projection, the projection plane, and f_x, r_1, t_1 and t_3 as distances in camera/screen space.

To be clear, $\phi(\mathbf{t})$ is non-affine because it involves a division by t_3 and an application of the ℓ_2 norm. Because ϕ is non-affine, we will approximate it with an affine Taylor series, so we can use equation 2.16. We define the *local affine approximation* of ϕ with the first two terms of its Taylor expansion around the point $\mathbf{t}_i$:

$$\phi_i(\mathbf{t}) \approx \phi(\mathbf{t}_i) + \mathbf{J}_i \cdot (\mathbf{t} - \mathbf{t}_i) \tag{2.18}$$

where $\mathbf{t}_i = \varphi(\boldsymbol{\mu}_i)$ is the center of the ith Gaussian in camera space, $\phi(\mathbf{t}_i)$ is the same point mapped to ray space, and $\phi_i(\mathbf{t}_i) = \phi(\mathbf{t}_i) = \hat{\boldsymbol{\mu}}_i$. The Jacobian of ϕ is defined with the matrix

$$\mathbf{J}_i = \left[\frac{\partial \phi}{\partial t_{i,1}}, \frac{\partial \phi}{\partial t_{i,2}}, \frac{\partial \phi}{\partial t_{i,3}}\right] = \begin{bmatrix} f_x/t_{i,3} & 0 & -f_x \cdot t_{i,1}/t_{i,3}^2 \\ 0 & f_y/t_{i,3} & -f_y \cdot t_{i,2}/t_{i,3}^2 \\ t_{i,1}/\ell_i & t_{i,2}/\ell_i & t_{i,3}/\ell_i \end{bmatrix} \tag{2.19}$$

where $\ell_i = ||(t_{i,1}, t_{i,2}, t_{i,3})^T||$[31]. Now we can approximate $\phi(\varphi(\mathbf{x}))$ with $m_i(\mathbf{x}) = \phi_i(\varphi(\mathbf{x}))$. Substituting $\mathbf{t} = \varphi(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{d}$ into equation 2.18, we have

$$\begin{aligned} \phi_i(\varphi(\mathbf{x})) \approx m_i(\mathbf{x}) &= \phi(\mathbf{t}_i) + \mathbf{J}_i \cdot (\mathbf{W}\mathbf{x} + \mathbf{d} - \mathbf{t}_i) \\ &= \mathbf{J}_i\mathbf{W}\mathbf{x} + \mathbf{J}_i(\mathbf{d} - \mathbf{t}_i) + \phi(\mathbf{t}_i). \end{aligned} \tag{2.20}$$

Combining equations 2.15 and 2.16, we have

$$\begin{aligned} g_i'(\mathbf{r}) &= g_i(m_i^{-1}(\mathbf{r})) && \text{definition of } g' \\ &= \mathcal{G}^3_{\Sigma_i}(m_i^{-1}(\mathbf{r}) - \boldsymbol{\mu}_i) && \text{equation 2.15} \\ &\approx \frac{1}{|\mathbf{M}^{-1}|}\mathcal{G}^3_{\mathbf{M}\Sigma_i\mathbf{M}^T}(\mathbf{r} - m_i(\boldsymbol{\mu}_i)) && \text{equation 2.16} \qquad (2.21) \\ &= \frac{1}{|\mathbf{M}^{-1}|}\mathcal{G}^3_{\mathbf{M}\Sigma_i\mathbf{M}^T}(\mathbf{r} - \boldsymbol{\mu}_i') \qquad (2.22) \end{aligned}$$

where $\mathbf{M} = \mathbf{J}_i\mathbf{W}$ by equation 2.20 [31]. To summarize our findings, line 2.22 can be rewritten to the form

$$g'(\mathbf{r}) = \frac{1}{|\mathbf{M}^{-1}|}\mathcal{G}^3_{\Sigma_i'}(\mathbf{r} - \boldsymbol{\mu}_i'))$$

where $\Sigma_i' = (\mathbf{J}_i\mathbf{W})\Sigma_i(\mathbf{J}_i\mathbf{W})^T$ and $\boldsymbol{\mu}_i' = m_i(\boldsymbol{\mu}_i) = \phi(\varphi(\boldsymbol{\mu}_i))$.

2.5.3 Projecting Gaussians From Ray Space to Screen Space

Equation 2.11 defines $\hat{g}_i(\mathbf{p}) = \int_0^\infty g_i'(\mathbf{p}, \xi)d\xi$ which effectively projects the ray space Gaussian g_i' to screen space. This subsection will define the covariance matrix and mean of the screen space Gaussian $\hat{g}_i$.

Section 2.5.2 provides definitions for Σ_i' and $\boldsymbol{\mu}_i'$ which can be used as a starting point for constructing the definitions of $\hat{\Sigma}_i$ and $\hat{\boldsymbol{\mu}}_i$. The explanation of ray space from section 2.3 indicates that $\boldsymbol{\mu}_i' = (\hat{\boldsymbol{\mu}}_i, \mu_{i,3})$ because projecting points from ray space is accomplished through ignoring the depth component. A similar principle can be used to define the screen space covariance matrix. In the ray space covariance matrix for $g_i'(\mathbf{r})$,

$$\Sigma_i' = \begin{bmatrix} s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix},$$

the last row and column correspond to the covariance involving the depth axis. Ignoring the last row and column of Σ_i' yields the covariance matrix of the corresponding screen space Gaussian, $\hat{g}_i(\mathbf{p})$[14]:

$$\hat{\Sigma}_i = \begin{bmatrix} s_{1,1} & s_{1,2} \\ s_{2,1} & s_{2,2} \end{bmatrix}.$$

These observations directly yield algorithm 3, which must be run as prerequisite to rendering.

2.5.4 Preprocess and Rasterization

To use algorithm 1 effectively, the list of Gaussians, $G = \{\hat{g}_1, \ldots, \hat{g}_n\}$, needs to be sorted so that a given pixel is contained within the support of every Gaussian with index $i_{\text{start}}(\mathbf{p}) \leq i < i_{\text{final}}(\mathbf{p})$. This sorting is accomplished with a tile-based approach. The screen is first divided into tiles of 16×16 pixels

Algorithm 2 Gaussian Covariance Projection[14].

This algorithm projects the covariance matrix of the ith Gaussian from world space to screen space.

Require: world space mean: $\boldsymbol{\mu}_i = (\mu_{i,0}, \mu_{i,1}, \mu_{i,2})^T$
world space covariance matrix: Σ_i
rotation matrix: $\mathbf{W}$
translation vector: $\mathbf{d}$
focal lengths: f_x and f_y

1: $(t_{i,1}, t_{i,2}, t_{i,3})^T \leftarrow \mathbf{W}\boldsymbol{\mu}_i + \mathbf{d}$
2: $\ell_i \leftarrow ||(t_{i,1}, t_{i,2}, t_{i,3})^T||$
3: $\mathbf{J_i} \leftarrow \begin{bmatrix} f_x/t_{i,3} & 0 & -f_x \cdot t_{i,1}/t_{i,3}^2 \\ 0 & f_y/t_{i,3} & -f_y \cdot t_{i,2}/t_{i,3}^2 \\ t_{i,1}/\ell_i & t_{i,2}/\ell_i & t_{i,3}/\ell_i \end{bmatrix}$ ▷ equation 2.19
4: $\mathbf{M} \leftarrow \mathbf{J_i}\mathbf{W}$
5: $\begin{bmatrix} s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} \leftarrow \mathbf{M}\Sigma_i\mathbf{M}^T$ ▷ equation 2.22
6: $\hat{\Sigma}_i \leftarrow \begin{bmatrix} s_{1,1} & s_{1,2} \\ s_{2,1} & s_{2,2} \end{bmatrix}$
7: **return** $\hat{\Sigma}_i$

and then copies of the Gaussians in G are assigned such that if the support of $\hat{g}_i$ might intersect tile t_j, then $\hat{g}_i \in t_j$. Within each tile, each member Gaussian is sorted by its ray space depth (recall that equation 2.13 is only valid if the Gaussians are sorted by depth). By concatenating the Gaussians in each consecutive tile, we are left with G': a master list of Gaussians hierarchically sorted by tile index and then depth. Note that copies of an individual Gaussian may appear multiple times in G' as it may intersect multiple tiles. In practice, this sort is performed with a single GPU radix sort of the Gaussians on a 64-bit key, where the upper 32 bits represent the tile index and the lower 32 bits represent depth[14].

Recall the definition of 3D Gaussians given in equation 2.7:

$$\mathcal{G}^3_\Sigma(\mathbf{x} - \boldsymbol{\mu}) = \frac{1}{(2\pi)^{3/2}|\Sigma|^{1/2}} e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu})}.$$

The world space covariance matrix of the ith Gaussian, Σ_i, is an optimizable scene parameter, but it must remain positive definite at each training iteration to avoid degenerate Gaussians. More precisely, the conic Σ_i^{-1} must remain positive definite, but since positive definiteness is closed under inversion, if Σ_i is positive definite, then Σ_i^{-1} must be as well. There is no easy way to constrain Σ_i during gradient descent, so an alternative representation is used. Instead of storing matrix values for Σ_i directly, the shape of the Gaussian is expressed as a scaling matrix $\mathbf{S}_i$ and rotation matrix $\mathbf{R}_i$[14]. The corresponding covariance matrix is then recovered with

$$\Sigma_i = (\mathbf{R}_i\mathbf{S}_i)(\mathbf{S}_i\mathbf{R}_i)^T. \tag{2.23}$$

To further simplify the training of Σ_i, the rotation matrix $\mathbf{R}$ is stored as a quaternion $\mathbf{q}_i$ and the scaling matrix $\mathbf{S}$ is stored as a scaling vector $\mathbf{s}_i$. Quaternions are four-element vectors that encode rotation[14].

To facilitate the full projection pipeline including the calculation of each covariance matrix from its constituent parts, we define a preprocess function that must be run for each Gaussian before scene rasterization. Algorithm 3 projects the mean of the ith Gaussian to screen space, computes the inverse of its screen space covariance matrix, finds the principal radii of its local support in screen space, and calculates how many tiles it could potentially overlap.

Rasterization is then performed according to algorithm 4.

2.6 3DGS Optimization

While the 3DGS renderer described previously in this chapter is fully differentiable, it cannot be used naively to reconstruct a scene as described in figure 2.1. In reality, there are several nuances that an effective training scheme must account for.

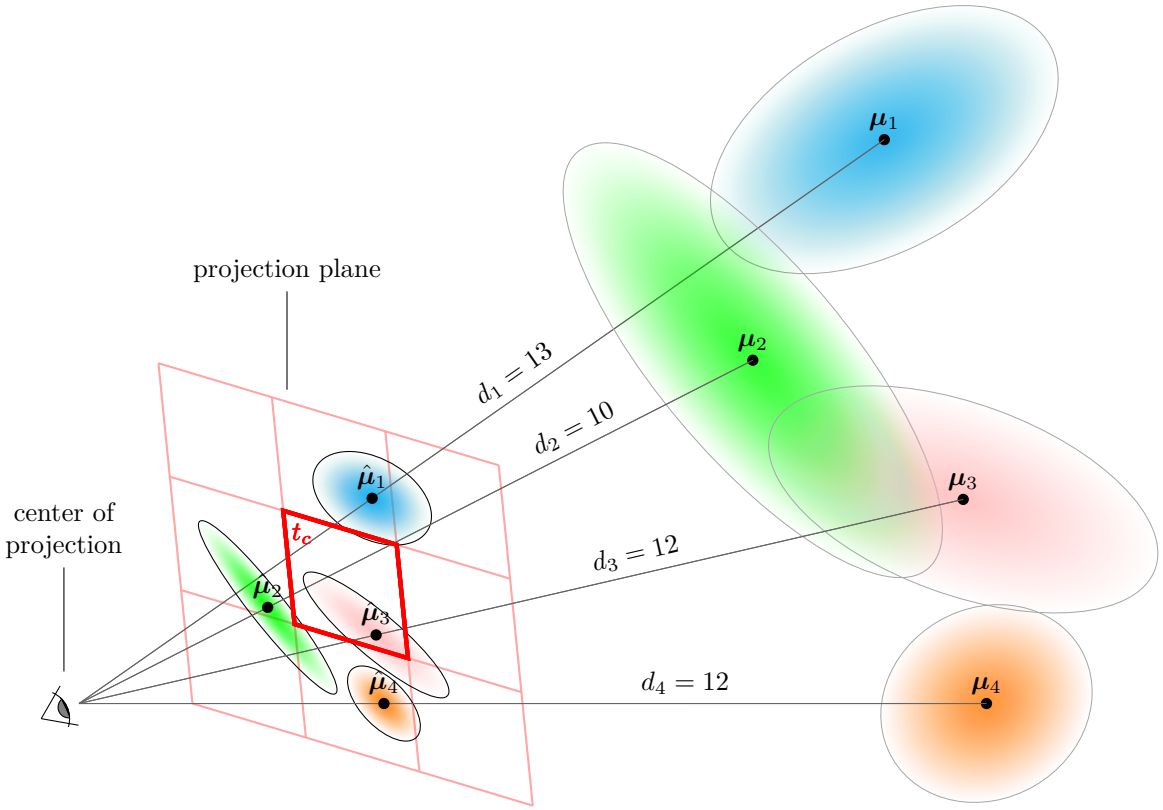

Figure 2.6: **Tile rendering**. Gaussians g_1, g_2, g_3, and g_4 with means $\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \boldsymbol{\mu}_3$ and $\boldsymbol{\mu}_4$ respectively are projected from world space to screen space. Although each Gaussian extends infinitely in all directions, we have clipped each Gaussian's domain to include only the *local support* region, shown as a thin gray ellipse surrounding the densest portion of each Gaussian. Any point outside of the local support region of a Gaussian will evaluate to zero. The projected Gaussians $\hat{g}_1, \hat{g}_2, \hat{g}_3$ and $\hat{g}_4$ with means $\hat{\boldsymbol{\mu}}_1, \hat{\boldsymbol{\mu}}_2, \hat{\boldsymbol{\mu}}_3$ and $\hat{\boldsymbol{\mu}}_4$ respectively also have their local support regions shown as dark gray ellipses on the projection plane. Surrounding the local support region of each projected Gaussian, the region's bounding box is shown as a blue rectangle.

The projection plane has been divided into a grid of individual tiles shown in light red. The center tile t_c is shaded darkly in red to indicate our area of interest. Gaussians $\hat{g}_3, \hat{g}_2$ and $\hat{g}_1$ are members of t_c because their bounding boxes intersect it whereas the bounding box of $\hat{g}_4$ does not. Moreover, Gaussians $\hat{g}_2, \hat{g}_3$ and $\hat{g}_1$ are sorted under t_c in that order because g_2 is closest to the center of projection with a distance of $d_2 = 10$, followed by g_3 and g_1 with $d_3 = 12$ and $d_1 = 13$ respectively. To render any pixel within t_c, algorithm 1 will evaluate $\hat{g}_3, \hat{g}_2$ and $\hat{g}_1$ in that order due to their hierarchical ordering under t_c. For a better view of the projection plane, please see figure 2.7.

Please note that figure 2.6 is an artistic interpretation of the Gaussian projection process. Values, distances, and shapes are inspired by mathematical results but optimized for visual clarity and legibility. In particular, the projection plane has been greatly magnified, the local linear approximation normally used to project Gaussians has been replaced with an arbitrary affine transformation, and the appearance of each Gaussian is approximated with dozens of translucent ellipses with different radii. To experiment with how the projection operation actually looks, readers are invited to tinker with the test script that inspired this diagram, available at https://github.com/UncatchableAlex/Thesis_Diagrams/blob/main/gproj.py

.

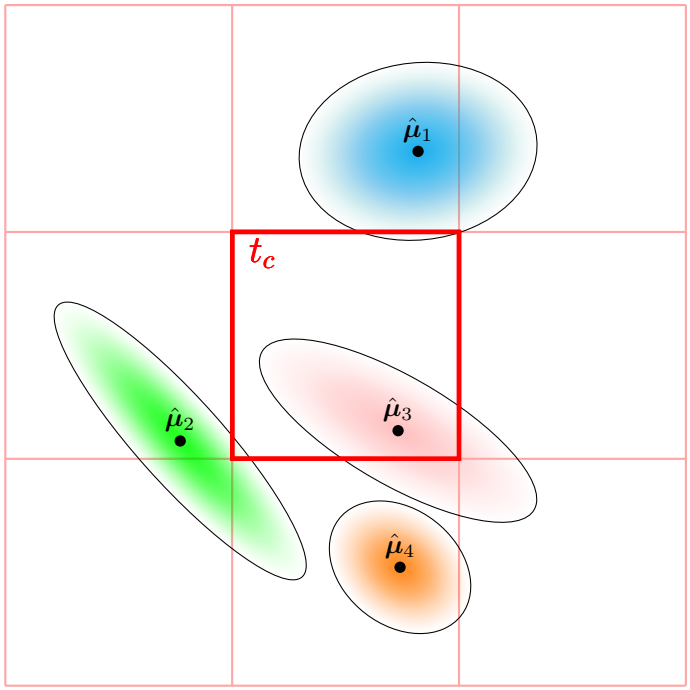

Figure 2.7: **Tile rendering (projection plane view)**. This is a close-up view of the projection plane from figure 2.6

Algorithm 3 Preprocess

Require: rotation matrix: $\mathbf{W}$
translation vector: $\mathbf{d}$
world space mean: $\boldsymbol{\mu}_i$
quaternion: $\mathbf{q}_i$
scaling factor: $\mathbf{s}_i$
focal length for x direction: f_x
focal length for y direction: f_y
width (in pixels) w
height (in pixels) h

$\mathbf{t}_i \leftarrow \mathbf{W}\boldsymbol{\mu}_i + \mathbf{d}$
$\hat{\boldsymbol{\mu}} \leftarrow ((w/2) + f_x \mathbf{t}_{i,1}/\mathbf{t}_{i,3}, (h/2) + f_y \mathbf{t}_{i,2}/\mathbf{t}_{i,3})^T$ ▷ The standard ray space projection
▷ with "pixel" translation applied
depth $\leftarrow \mathbf{t}_{i,3}$
if $\hat{\boldsymbol{\mu}}$ is off-screen **then**
  Return Invalid Gaussian
end if
$\Sigma \leftarrow$ **compute_cov_3D**$(\mathbf{q}_i, \mathbf{s}_i)$ ▷ Algorithm not shown. Refer to section 2.5.4 for details
$\hat{\Sigma} \leftarrow$ **compute_cov_2D**$(\hat{\boldsymbol{\mu}}, \Sigma)$ ▷ algorithm 2
$d \leftarrow \det(\hat{\Sigma})$
$\hat{\Sigma}^{-1} \leftarrow \begin{bmatrix} \frac{\Sigma_{0,0}}{d} & -\frac{\Sigma_{0,1}}{d} \\ -\frac{\Sigma_{1,0}}{d} & \frac{\Sigma_{1,1}}{d} \end{bmatrix}$ ▷ 2×2 matrix inverse trick
$\lambda_1, \lambda_2 \leftarrow$ **eigenvalues**$(\hat{\Sigma})$
$r \leftarrow 3\sqrt{\max(\lambda_1, \lambda_2)}$ ▷ three standard deviations
box $\leftarrow$**get_rect_2d**$(\boldsymbol{\mu}, r)$ ▷ get the bounding box of the support in tile space
num_tiles $\leftarrow$(box.xhi - box.xlo) $\times$ (box.yhi - box.ylo)
return $\hat{\boldsymbol{\mu}}, \hat{\Sigma}^{-1}$, num_tiles, box, r, depth

Algorithm 4 Rasterize

Require: $n > 0$, $\epsilon > 0$
Opacities: $\sigma_1, \ldots, \sigma_n$
Colors: $\mathbf{c}_1, \ldots, \mathbf{c}_n$ where $\mathbf{c}_i = (c_{i,\lambda_0}, \ldots, c_{i,\lambda_m})^T$
Screen-space means: $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_n$
Quaternions: $\mathbf{q}_1, \ldots, \mathbf{q}_n$
Scale vectors: $\mathbf{s}_1, \ldots, \mathbf{s}_n$
background color C_{bg}
rotation matrix: $\mathbf{W}$
translation vector: $\mathbf{d}$
focal length for x direction: f_x
focal length for y direction: f_y
width (in pixels) w
height (in pixels) h

$G \leftarrow [\ ]$ ▷ assemble the list of projected Gaussians
for $i \in \{1, \ldots, n\}$ **do**
  $G \leftarrow G \oplus \texttt{preprocess}(\mathbf{W}, \mathbf{d}, \boldsymbol{\mu}_i, \mathbf{q}_i, \mathbf{s}_i, f_x, f_y, w, h)$ ▷ see algorithm 3
end for
$G' \leftarrow \texttt{key\_value\_list\_by\_tile}(G)$ ▷ pseudocode not provided. Described in section 2.5.4
$G' \leftarrow \texttt{radix\_sort}(G')$
$\mathbf{I} \leftarrow \mathbf{0}$ ▷ initialize the pixel array canvas
for $\mathbf{p} \in \texttt{indices}(\mathbf{I})$ **do** ▷ parallel loop
  $\mathbf{I}_\mathbf{p} \leftarrow \texttt{pixel\_color}(G', \mathbf{p})$ ▷ calculate the color of every pixel
end for
return I

2.6.1 Activation Functions

Recall that the scene representation used by 3DGS is a list of 3D Gaussian functions defined by a world space mean $\boldsymbol{\mu}_i$, scale vector $\mathbf{s}_i$, and quaternion $\mathbf{q}_i$, augmented with a spherical harmonic coefficient color vector $\mathbf{c}_i$ and opacity σ_i. Or, more concisely, $\theta = \{(\boldsymbol{\mu}_j, \mathbf{s}_j, \mathbf{q}_j, \mathbf{c}_j, \sigma_j)\}_{j=1}^n$ where each parameter for each Gaussian must receive gradients from the renderer with each training iteration.

To constrain each σ_j to $(0, 1)$, Kerbl et al. use a sigmoid activation function. Similarly, they also use an exponential activation function for each $\mathbf{s}_j$[14]. All other parameters use a linear activation.

2.6.2 Loss Function

Kerbl et al. use the loss function

$$\mathcal{L} = (1 - \lambda)\mathcal{L}_1 + \lambda\mathcal{L}_{\text{D-SSIM}}$$

where

$$\mathcal{L}_1(\mathbf{I}, \mathbf{I}^*) = \frac{1}{|\mathbf{I}|} \sum_{p \in \texttt{indices}(\mathbf{I})} |\mathbf{I}_p - \mathbf{I}_p^*|$$

represents mean absolute error, and $\mathcal{L}_{\text{D-SSIM}}(\mathbf{I}, \mathbf{I}^*) = \text{D-SSIM}(\mathbf{I}, \mathbf{I}^*)$ is the dissimilarity version of the Structural Similarity Index Measure (SSIM), defined as $\text{D-SSIM}(\mathbf{I}, \mathbf{I}^*) = \frac{1 - \text{SSIM}(\mathbf{I}, \mathbf{I}^*)}{2}$ by Wang et al.[28]. Empirically, Kerbl et al. chose $\lambda = 0.2$.

2.6.3 Initialization

Because the input to 3DGS training is a dataset of unlabeled images, the first step is to estimate the camera pose associated with each. For this task, Kerbl et al. rely on Structure from Motion (SfM)[26], a previous project that estimates camera pose and reconstructs a sparse point cloud of the scene. The 3DGS scene is then initialized from this point cloud, with each reconstructed point becoming the mean of a Gaussian, $\boldsymbol{\mu}_i$.

To initialize the covariance matrices of the Gaussians, Kerbl et al. use

$$\Sigma_i = \begin{bmatrix} d_i^2 & 0 & 0 \\ 0 & d_i^2 & 0 \\ 0 & 0 & d_i^2 \end{bmatrix}$$

where d_i is the average distance from the ith point to the closest three other points in the cloud[14].

2.6.4 Warm-up

Kerbl et al. note that the first spherical harmonic band representing the "base" color has a tendency to train poorly in scenes where some angular regions are missing (for example, if the scene is the inside of a room with corners). To facilitate effective training of the spherical harmonics, only the first band is trained for the first 1000 iterations, and one band is added to the training every 1000 iterations thereafter until each Gaussian's color is represented with four bands.

2.6.5 Learning Rates

All learning rates in 3DGS are constant except for $\boldsymbol{\mu}_i$, which uses an exponential decay[14]. As training progresses, the world space position of each Gaussian becomes increasingly resistant to change, and the optimization process focuses increasingly on color, opacity, and the shape of the Gaussians.

2.6.6 Densification and Opacity Culling

During the training process, two malignant processes can prevent the accurate reconstruction of local scene details. The first issue, *under-reconstruction*, occurs when areas of the scene are not covered by Gaussians, resulting in bare spots where geometric detail is missing. To address under-reconstruction, 3DGS training occasionally clones each Gaussian in under-reconstructed areas, meaning that an exact copy of each Gaussian is added to the scene, shifted in the direction of the original Gaussian's positional gradient[14].

In the other case, *over-reconstruction*, fine local detail in the scene is obscured by large Gaussians. In these areas, Gaussians need to be split, meaning that each large Gaussian is split into two smaller ones. The position of these two new Gaussians is determined using the original Gaussian as a probability density function. The size of the two new Gaussians is determined by dividing the scale vector of the original Gaussian by $\phi = 1.6$, a factor determined experimentally by Kerbl et al. Together, cloning and splitting comprise the 3DGS *densification* protocol [14].

Both over- and under-reconstructed areas exhibit Gaussians with large positional gradients in screen space. In 3DGS, a Gaussian undergoes densification if the average gradient for $\hat{\boldsymbol{\mu}}_i$ exceeds a threshold value $\tau_{\text{pos}} = 0.0002$. Furthermore, a Gaussian undergoes splitting if its maximum scale exceeds a fraction of the scene extent, indicating that the Gaussian covers a large spatial region and should be subdivided into smaller Gaussians. By default, this fraction is set to $\tau_{\text{ext}} = 0.01$. Gaussians whose average gradient for $\hat{\boldsymbol{\mu}}$ exceeds τ_{pos}, but whose largest scale does not exceed τ_{ext}, undergo cloning[14].

By default, scene densification runs every 100 training iterations along with an opacity culling procedure where any Gaussian with $\sigma_i < \epsilon_\sigma$ is removed from the scene [14].

2.6.7 Opacity Reset

Occasionally throughout training, a stray Gaussian may become positioned inappropriately far away from the surface that it is meant to comprise. These stray Gaussians are referred to by Kerbl et al. as *floaters*, probably due to their appearance as floating blobs hovering throughout the scene. To help limit their appearance, the opacity of all Gaussians is reset to a value close to zero every 3000 iterations. This technique, coupled with the opacity culling mentioned in the previous section, effectively removes floaters.

2.6.8 Optimizer

To accelerate training, Adam[15] is used to determine the step size at each training step.

2.7 Differentiation Approaches

Although a differentiable renderer guarantees the possibility of gradient calculation, the actual mechanism by which this calculation occurs remains a matter of choice. We will cover the four common approaches for determining how the outputs of a computer program vary with respect to its inputs: numerical, symbolic, automatic, and manual differentiation. Included is a discussion of the trade-off between automatic and manual differentiation, and a conclusion with a worked-through example of the implementation of a derivative for algorithm 1 from Kerbl et al.[14].

2.7.1 Numerical Differentiation

Numerical differentiation approximates a derivative by evaluating the function at perturbed inputs, and thereby introduces both approximation errors and floating point errors to the result[10]. For example, from a multivariate function $f : \mathbb{R}^n \to \mathbb{R}$, one may approximate the gradient $\nabla f(\mathbf{x}) = [\frac{\partial f}{\partial x_1}(\mathbf{x}), \ldots, \frac{\partial f}{\partial x_n}(\mathbf{x})]^T$ with the *forward difference method*

$$\frac{\partial f}{\partial x_i}(\mathbf{x}) \approx \frac{f(\mathbf{x} + h\mathbf{e_i}) - f(\mathbf{x})}{h} + O(h)$$

where $h > 0$ is the step size in the direction of $\mathbf{e_i}$, and $O(h)$ is the approximation error[2]. Approximation error decreases with h, but floating point errors become increasingly problematic. Other, more accurate approximations exist for $\frac{\partial f}{\partial x_i}$ that require multiple applications of f such as the *center difference method*

$$\frac{\partial f}{\partial x_i}(\mathbf{x}) \approx \frac{f(\mathbf{x} + h\mathbf{e_i}) - f(\mathbf{x} - h\mathbf{e_i})}{h} + O(h^2).$$

Such approaches can facilitate the use of a larger h to ease floating point errors but fundamentally encounter a trade-off between accuracy and performance.

Worse still, numerical differentiation scales poorly for large n as each partial derivative must be computed independently. For example, one application of the center difference method will approximate $\frac{\partial f}{\partial x_1}$, but another application will be needed to approximate $\frac{\partial f}{\partial x_2}$. In the case of Gaussian splatting, the input to the rasterizer may be 10^6 Gaussians each with 14 parameters[9], implying that a single training iteration would require 14×10^6 applications of the center difference method. Because each application of the center difference method requires 2 function calls, numerical methods could require the scene to be rasterized 28 million times for a single training iteration of 3DGS.

2.7.2 Symbolic Differentiation

All the faults of numerical differentiation lie in the way it uses discrete values to approximate infinitesimals. To overcome this barrier, every other differentiation approach opts for mathematically correct calculus.

Symbolic differentiation calculates an exact gradient by computing and evaluating the analytical derivative of a closed-form function. Unfortunately, most code relies on branching constructs which produce execution-dependent derivatives that cannot be easily differentiated as closed-form expressions. The fundamental issue occurs at runtime when a branch is encountered, and execution resumes down one path, making the final function result dependent on that particular branching decision. A closed-form symbolic expression cannot represent this behavior on its own because it maintains no record of which branch was taken during execution. As such, native symbolic differentiation approaches are forced to differentiate *every* branch of a piecewise function—a ruinous waste of resources if many branches are present.

2.7.3 Forward Mode Automatic Differentiation

Automatic differentiation extends the principles of naive symbolic differentiation by splitting a function, f, into its constituent steps, which can then be differentiated individually. These individual derivatives can then be used to track the gradient flow through f via the chain rule.

[9] We assume an RGB color scheme here. 3DGS usually uses spherical harmonics which become larger as training progresses.

AD's first variant, forward mode, is the simplest and easiest to understand. Given a primal function, $f : \mathbb{R}^n \to \mathbb{R}^m$, with inputs $x_1, \ldots, x_n$ and outputs $y_1, \ldots, y_m$, we seek to calculate the *Jacobian-vector product* (JVP)

$$\mathbf{J}_f \mathbf{r} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \Bigg|_{\mathbf{x}=\mathbf{a}} \begin{bmatrix} r_1 \\ \vdots \\ r_n \end{bmatrix}$$

where $\mathbf{a}$ is the point at which the JVP of f is evaluated. Most commonly, we will use the JVP to calculate the jth column of f's Jacobian, $\mathbf{J}_f^{(j)} = [\frac{\partial y_1}{\partial x_j} \ldots \frac{\partial y_m}{\partial x_j}]^T$ which can be done by setting $\mathbf{r} = \mathbf{e}_j$.

Forward mode AD works by executing each calculation step, v_i, as normal, while simultaneously tracking a separate *forward tangent* value, $\dot{v}_i = \frac{\partial v_i}{\partial x_j}$[2]. We demonstrate this behavior in figure 2.8 by using forward mode AD to compute $\frac{\partial f}{\partial x_1}$ for the function $f(x_1, x_2) = \ln(x_1^2 + x_2^2) + x_1^3$. Notice how we initialize the tangent trace with $\dot{x}_1 = 1$ and $\dot{x}_2 = 0$ to calculate $\frac{\partial f}{\partial x_1}$. If we wanted to calculate $\frac{\partial f}{\partial x_2}$, we would initialize the tangent trace with $\dot{x}_1 = 0$ and $\dot{x}_2 = 1$.

The effectiveness of forward mode AD depends largely on the shape of $\mathbf{J}_f$. Because the JVP may only supply a single column of $\mathbf{J}_f$ at a time, we are obligated to run the algorithm n times to reconstruct the full Jacobian. For functions like 3DGS rasterization, where $n \gg m$, forward mode AD immediately succumbs to the same fault as numerical differentiation: In order to calculate the sensitivity of every input with respect to a single output, forward mode must be run n times. To use forward mode AD to train a 3DGS scene containing 10^6 Gaussians each with 14 features, we would have to run 14 million AD traces to complete a single training iteration!

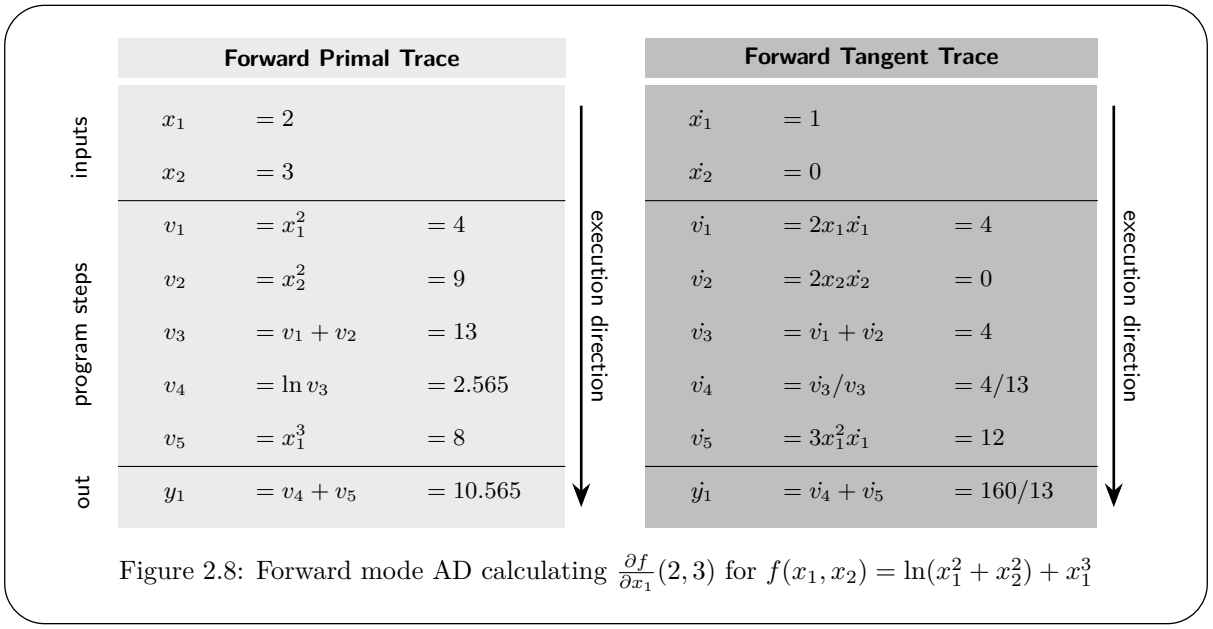

Figure 2.8: Forward mode AD calculating $\frac{\partial f}{\partial x_1}(2, 3)$ for $f(x_1, x_2) = \ln(x_1^2 + x_2^2) + x_1^3$

2.7.4 Reverse Mode Automatic Differentiation

Unlike forward mode AD, which computes a column of $\mathbf{J}_f$ through a JVP, reverse mode AD computes a row of $\mathbf{J}_f$ through a *vector-Jacobian product* (VJP)

$$\mathbf{b}^T \mathbf{J}_f = \begin{bmatrix} b_1, \ldots, b_m \end{bmatrix} \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \Bigg|_{\mathbf{x}=\mathbf{a}}$$

The ith row of $\mathbf{J}_f$ can be obtained by setting $\mathbf{b} = e^i$. For functions $f : \mathbb{R}^n \to \mathbb{R}^m$ where $m \ll n$, reverse mode AD can provide significantly more information than a single application of forward mode.

The principle of reverse mode is to work backward, calculating the derivative of a single output with respect to each intermediate variable. Reverse mode works in two passes: a *forward primal trace* which

executes the function in the normal "forward" direction, and a *backward adjoint trace* which calculates the *adjoint* of each v_j, defined as

$$\overline{v_j} = \frac{\partial y_i}{\partial v_j}.$$

During the forward primal trace, each intermediate result (associated with each v_j) is stored in memory on a data structure known as the *tape* for use during the backward adjoint trace. Additionally, the forward primal trace generates a dependency graph, D, of the intermediate steps (shown in figure 2.10).

We can then perform the backward adjoint trace by calculating each $\overline{v_j}$ in reverse order using

$$\overline{v_j} = \frac{\partial y_i}{\partial v_j} = \sum_{v_k \in D(v_j)} \overline{v_k} \frac{\partial v_k}{\partial v_j} \tag{2.24}$$

where $D(v_j)$ is the set of steps in f that depend on v_j, and $\frac{\partial v_k}{\partial v_j}$ is an analytical derivative calculated from v_k[2]. Interestingly, equation 2.24 is simply the multivariable chain rule applied recursively over the dependency graph D. Since the adjoint trace traverses D in reverse order, the adjoint of every $v_k \in D(v_j)$ has already been computed when $\overline{v_j}$ is evaluated. Also, note that although figure 2.9 separates inputs, program steps, and outputs for clarity, all three classes are intermediate steps as we define them in this section. For example, in the context of figure 2.9, the input x_2 can be thought of as intermediate step v_0, and the output y_1 is an alias of the intermediate step v_6.

Reverse mode differentiation is the method of AD best suited for finding gradients of the loss functions used in scene reconstruction and will remain a central topic for the rest of this paper. Loss functions accept many inputs and return a single scalar loss, implying that a single application of reverse mode AD is sufficient for calculating the complete Jacobian of the rasterizer.

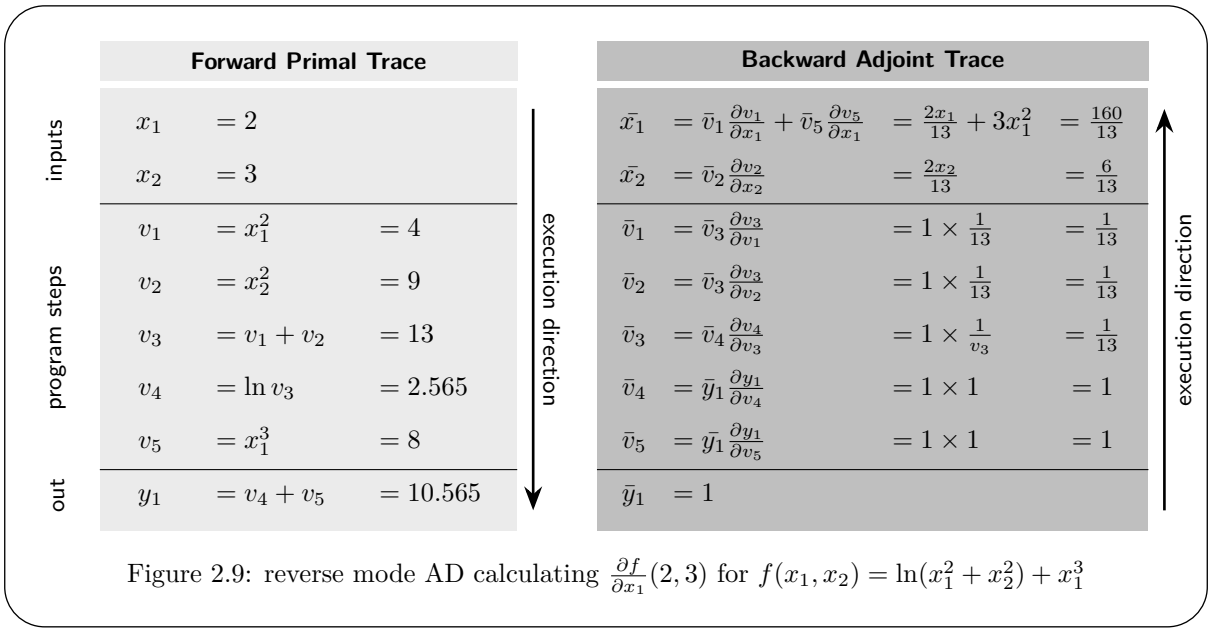

Figure 2.9: reverse mode AD calculating $\frac{\partial f}{\partial x_1}(2,3)$ for $f(x_1, x_2) = \ln(x_1^2 + x_2^2) + x_1^3$

2.7.5 Comparing Manual Differentiation and AD

Manual differentiation is the process of hand-coding a discrete gradient function to calculate selected derivatives needed by the programmer. Manual differentiation is ad hoc in nature and therefore difficult to classify in general terms. However, writing a bespoke gradient function affords the programmer a much greater degree of control over the derivative calculation and can therefore be used to yield massive performance improvements in certain problem spaces.

In the case of 3DGS, manual differentiation exposes a variety of domain-specific optimizations that are inaccessible to AD and dramatically reduce running time. For example, reverse mode AD will record intermediate values throughout the primal trace of the radix sort step and subsequently differentiate each operation during the backward adjoint trace. An experienced programmer will recognize that the entire radix sort can be considered its own intermediate step in the AD algorithm, $v_i = \texttt{sort}(v_j)$, whose

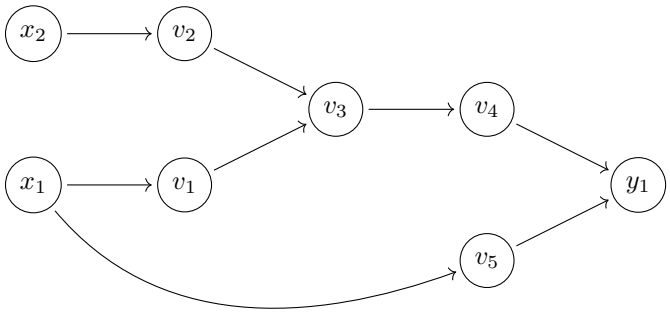

Figure 2.10: Primal Trace Dependency Tree From Figure 2.9

derivative during the backward trace simply permutes the incoming adjoints according to the inverse permutation calculated during sorting. Rather than differentiating each low-level operation within the sort function with AD, the programmer may directly implement the gradient propagation rule for radix sort.

Another optimization opportunity arises when calculating $\frac{\partial L}{\partial C}$ from equation 2.13. An efficient implementation (employing either manual or automatic differentiation) will calculate pixel color with a single for-loop over the outer summation. The inner product loop representing transmittance will be tracked through a separate accumulation variable, T, as seen in algorithm 1. Unfortunately, this looping approach will result in reverse mode AD storing every intermediate value of C and T on the tape. Because the loop is simple to calculate but difficult to store for large N, a more disciplined approach would save only the final value of T from the forward pass and recalculate intermediate values in reverse order using algorithm 5 (derived in section 4.4).

Although the original 3DGS authors[14] do a fantastic job identifying and implementing domain-specific optimizations, manual gradient functions can be unforgiving. A substantial amount of algebra may be required to statically determine gradient update rules within loops, and any mistake will propagate through the rest of the program. Furthermore, the programmer must have a firm understanding of where each step in their program is placed within the dependency hierarchy dictating gradient flow (see figure 4.1). If the programmer seeks to modify the primal function, they will likely be obligated to update the corresponding gradient function as well. Worse still, these updates to the gradient function may need to be applied to areas that do not directly correspond to the modified region of the primal function[8]. The effort of maintaining both the primal and gradient functions bloats the code base, effectively doubling the required labor of code improvement.

Algorithm 5 Pixel Color Gradient: This function returns this pixel's component of each Gaussian's gradient. See section 4.4 for the derivation of the gradients in this function.

Require: $n \geq 0$
Opacities: $\sigma_1, \ldots, \sigma_n$
Tristimulus functions: $\mathbf{c}_1, \ldots, \mathbf{c}_n$ where $\mathbf{c}_i = (c_{i,r}, c_{i,g}, c_{i,b})^T$
Forward pass outputs (from same pixel): $T_{\text{final}}, i_{\text{final}}$
Screen-space means: $\hat{\boldsymbol{\mu}}_1, \ldots, \hat{\boldsymbol{\mu}}_n$
2D Conics: $\hat{\Sigma}_1^{-1}, \ldots, \hat{\Sigma}_n^{-1}$
Pixel position $\mathbf{p}$
Index: i_{start} where $i_{\text{start}} \leq i_{\text{final}}$
Upstream gradient: $\overline{C}$
Viewing direction: $\mathbf{v}$

$T_{i_{\text{final}}+1} \leftarrow T_{\text{final}}$
$(r_r, r_g, r_b) \leftarrow (0, 0, 0)^T$
Initialize all output arrays, $\overline{\alpha_1}, \ldots, \overline{\alpha}_n$, $\overline{\sigma}_1, \ldots, \overline{\sigma}_n$, $\overline{\mathbf{c}}_1, \ldots, \overline{\mathbf{c}}_n$, $\overline{\hat{\Sigma}^{-1}}_1, \ldots, \overline{\hat{\Sigma}^{-1}}_n$, $\overline{\hat{\boldsymbol{\mu}}}_1, \ldots, \overline{\hat{\boldsymbol{\mu}}}_n$ to $\mathbf{0}$
for $i \in i_{\text{final}}, \ldots i_{\text{start}}$ **do**
    $\mathbf{d} \leftarrow \mathbf{p} - \hat{\boldsymbol{\mu}}_i$ ▷ Recover $\mathbf{d}$
    $\alpha_i \leftarrow \sigma_i \exp(-\frac{1}{2}\mathbf{d}^T \hat{\Sigma}_i^{-1} \mathbf{d})$ ▷ Recover α_i
    $T_i \leftarrow T_{i+1}/(1-\alpha_i)$
    $\overline{\alpha_i} \leftarrow 0$ ▷ Recover T_i
    for $k \in [r, g, b]$ **do**
        $\overline{\alpha_i} \leftarrow \overline{\alpha_i} + \overline{C_k}(c_{i,k}(\mathbf{v}) - r_k)$ ▷ Equation 4.16
        $\overline{c_{i,k}(\mathbf{v})} \leftarrow \overline{C_k}\alpha_i T_i$ ▷ Equation 4.8
        $r_k \leftarrow c_{i,k}(\mathbf{v})\alpha_i + (1-\alpha_i)r_k$ ▷ Equation 4.17
    end for
    $\overline{\alpha_i} \leftarrow T_i \overline{\alpha_i} - \frac{T_{\text{final}}}{(1-\alpha_i)}(C_{\text{bg}} \cdot \overline{C})$ ▷ Equation 4.16
    $\overline{\hat{\boldsymbol{\mu}}_i} \leftarrow \hat{\Sigma}_i^{-1} \mathbf{d} \overline{\alpha_i} \alpha_i$ ▷ Equation 4.11
    $\overline{\hat{\Sigma}_i^{-1}} \leftarrow \overline{\alpha_i}\alpha_i \begin{bmatrix} \frac{-d_1 d_1}{2} & -d_1 d_2 \\ -d_1 d_2 & \frac{-d_2 d_2}{2} \end{bmatrix}$ ▷ Equation 4.12
    $\overline{\sigma_i} \leftarrow \frac{\alpha_i \overline{\alpha_i}}{\sigma_i}$ ▷ Equation 4.14
end for
return $\overline{\alpha}, \overline{c}, \overline{\sigma}, \overline{\hat{\Sigma}^{-1}}, \overline{\hat{\boldsymbol{\mu}}}$

2.8 Futhark

Futhark[9] is a statically typed, purely functional array programming language with a compiler capable of generating parallel code for multi-threaded CPU and GPU backends, including CUDA and OpenCL. Futhark offers native support for forward mode and reverse mode automatic differentiation, making it an ideal candidate language for implementing 3DGS with AD.

Forward mode AD is achieved in Futhark with its **jvp** function. To demonstrate this capability, consider the following Futhark function which takes three 32-bit floats as input and produces a 32-bit float as output:

```
def f (a: f32) (b: f32) (c: f32): f32 =
    let inner = a**4 + b**3 + c**2
    in f32.sqrt inner
```

To calculate the Jacobian of **f**, we would need three invocations of **jvp**, one for each input of **f**. Written in Futhark, we have:

```
def forward (a: f32) (b: f32) (c:f32): (f32, f32, f32) =
    -- anonymous function taking a three-tuple as input and
    -- invoking f on its values
    let f' = \(a,b,c) -> f a b c

    -- define a three tuple from the inputs
    let x = (a,b,c)

    -- calculate jacobian
    let df_da = jvp f' x (1.0, 0, 0) -- df/da
    let df_db = jvp f' x (0, 1.0, 0) -- df/db
    let df_dc = jvp f' x (0, 0, 1.0) -- df/dc

    -- coalesce result
    in (df_da, df_db, df_dc)
```

Alternatively, we could accomplish the same operation with one invocation of **vjp**:

```
def reverse (a: f32) (b: f32) (c: f32) : (f32, f32, f32) =
    vjp (\(a, b, c) -> f a b c) (a, b, c) 1.0
```

Because of its native support for **vjp** and CUDA, Futhark is an ideal testbed for implementing a 3DGS inverse rendering system with AD. In the next chapter, such an implementation will be discussed.

Chapter 3

Experiments

3.1 Introduction

Although massively parallel implementations of 3DGS exist for the GPU, they typically rely on manual differentiation techniques that yield unwieldy code motivated by complicated math (see algorithm 5). As a relatively new language, Futhark's capabilities to automatically handle gradients in a massively-parallel graphics context remain untested. This chapter seeks to fill that research gap. Specifically, the question I wish to answer is:

Can automatic differentiation in Futhark replace the manually-derived gradients of a CUDA-based 3DGS renderer without incurring prohibitive performance cost?

By testing Futhark's capabilities in this context, I also hope to motivate further development of the Futhark compiler and provide real-world insight into its strengths and shortcomings.

Ultimately, I find that even though Futhark substantially simplifies the implementation, it comes with a significant performance cost relative to the manual gradient function supplied by the CUDA reference implementation. My findings suggest that AD is not yet a drop-in replacement for manual gradient functions in this setting.

In this chapter, I will cover the setup of the testing environment, the development of the Futhark implementation, a quantitative assessment of its performance, and a qualitative discussion of Futhark's merits and shortcomings in the context of 3DGS rendering[1].

3.2 Methods

The original 3DGS authors[14] provided their reference implementation in a GitHub repository listed on their website. My Futhark-enabled implementation was created from a fork of this repository. I identified the classes they used to interface their PyTorch[21] training loop with the CUDA rasterizer and replaced them with equivalent classes that interfaced with my Futhark implementation. In this section, I will summarize the original code provided by Kerbl et al., the changes I made to their code, the Futhark rasterizer I created, and how I tested it.

3.2.1 Starter Code

In structure, the code from Kerbl et al. involved a large Python training loop managing data loading, the spherical harmonic warm up, opacity culling, densification, and step sizes. A **`GaussianModel`** class stored the scene parameters as PyTorch[21] tensors on the GPU and exposed them to the training loop through their respective activation functions. For use within the training loop, the authors included a **`render`** function which, in turn, interacted with a **`_RasterizeGaussians`**[2] class that implemented the PyTorch AD interface, **`torch.autograd.Function`**. The **`_RasterizeGaussians`** class served to wrap their custom CUDA 3DGS rasterizer which included functions for normal rendering operation

[1] All my code for this project can be found at https://github.com/UncatchableAlex/gaussian-splatting

[2] A *rasterizer* is the part of the renderer which converts a vectorized representation (in this case, a list of scene parameters) into an array of pixels. In the context of this paper, the words renderer and rasterizer are somewhat interchangeable.

and manual gradient calculation. In effect, their **_RasterizeGaussians** allowed their manual gradient calculation in CUDA to integrate cleanly as part of PyTorch's built-in AD pipeline.

At the start of each iteration within their training loop, it first checks whether another spherical harmonic band should be added to each Gaussian's color representation. Next, a random image is selected from the ground truth data set along with its associated camera and fed into the **render** function. The loss value of the subsequently outputted image is then calculated using vectorized PyTorch operations. Because the entire rendering pipeline operates on PyTorch tensors and functions, PyTorch can then automatically propagate gradients backward through the loss function, the **render** function, the **_RasterizeGaussians** class, and the activation functions specified within the **GaussianModel** class. Next, the loop checks whether densification should be performed and appropriately adjusts the Adam optimizer moments for the added and deleted Gaussians. After checking for densification, which happens every 100 iterations by default, the loop checks if it is time for an opacity reset and does so if necessary. By default, the opacity reset only happens every 3000 iterations. Finally, the scene parameters are updated according to the Adam optimizer before the loop resets.

The training script, which the training loop is part of, is designed to be called with command line arguments specifying training parameters such as the color of the background, the number of iterations to train for, and which iterations to save checkpoint files. The script also accepts flags named **--convert_SH_python** and **--convert_cov3D_python** which specify what type of input the rasterizer is capable of accepting. If **--convert_SH_python** is given, the spherical harmonic representation will be simplified in Python, and the resulting RGB values will be passed to the renderer instead of the raw spherical harmonic coefficients. Similarly, if the **convert_cov3D_python** flag is passed, the 3D covariance inputs will be simplified by Python into corresponding 2D screen space covariance matrices before being passed to the rasterizer.

3.2.2 Futhark Rasterizer

After forking the 3DGS repository, I used their custom CUDA rasterizer as inspiration for my own Futhark version. My finished implementation exposes three functions: **rasterize** for normal rendering, **grad_naive** for an unoptimized reverse mode AD gradient computation, and **grad** which performs AD more efficiently. I used *futhark-0.26.1* for all compiling.

Futhark Rasterize Function

The definition of **rasterize** is fairly simple, but it differs from the reference CUDA implementation in a few key ways. Firstly, the original CUDA version used an inputted projection matrix to send Gaussian means directly from world space to screen space, whereas my version uses focal lengths to project the means to screen space through an explicit conversion to ray space (equation 2.17). My implementation also does not take a scale modifier, which acts as a coefficient to the scale vectors of the Gaussians. In essence, the scale modifier is assumed to always be 1. Furthermore, my implementation only accepts RGB values instead of raw spherical harmonic coefficients as color input. The effect of this design choice is that my rasterizer does not take arguments for the camera position, the spherical harmonic degree, or the spherical harmonics themselves, and may only be trained with the **--convert_SH_python** flag enabled. Similarly, my implementation only accepts 3D covariance matrices, which implies that the **--convert_cov3D_python** flag must *not* be given. My implementation also does not accept an *antialiasing* boolean parameter because it only implements the simple low-pass filter used by the original authors and not the more advanced antialiasing techniques. Although these techniques are relatively straightforward to implement, they are considerably more difficult to explain and motivate, so I made the pragmatic choice to omit them from this project. Below, I give the Futhark implementation of **rasterize** with the variable names changed to match the math presented in this paper:

```
-- Our exposed rasterize function. In this function, we take the means, scales,
-- rotations, colors, and opacities of the 3D Gaussians, as well as the camera
-- parameters, and output the radii of all the Gaussians and the RGB image.
entry rasterize [n]
                (background: [3]f32) -- the color of the background
                (μ: [n][3]f32) -- world space means
                (c: [n][3]f32)  -- the RGB colors of the gaussians
                (σ: [n][1]f32) -- opacities
                (s: [n][3]f32)  -- scale vectors
                (q: [n][4]f32) -- vector of quaternion rotations
```

```
                (view_matrix: [4][4]f32) -- encodes rotation and translation. A combination
                                         -- of W and d from Algorithm 3
                (tan_fovx: f32) -- encodes focal (x) length
                (tan_fovy: f32) -- encodes focal (y) length
                (image_height: i64)  -- in pixels
                (image_width: i64) : -- in pixels
([n]i32, [image_height][image_width][3]f32) =

  let h = i32.i64 image_height
  let w = i32.i64 image_width

  -- Call the Futhark implementation of preprocess (algorithm 3).
  -- Return n array of 2D Gaussian records derived from the scene parameters
  let gaussians = compute2dGaussians μ c σ s q view_matrix tan_fovx tan_fovy h w

  -- rasterize the 2D Gaussians into a pixel array using the loop cutoff (see section 3.2.2)
  -- This step is equivalent to "forward_pixel_color" (algorithm 1)
  in rasterize2dGaussians gaussians background image_height image_width false
```

Ultimately, **rasterize** returns a list of 32-bit integers representing the radius of each Gaussian and a 3D array representing every RGB pixel value. In this context, radius is defined as the distance corresponding to three standard deviations from the mean on the larger principal axis of a Gaussian function.

For the Gaussian sorting step, I used the blocked radix sort implementation from the Futhark sorting library[5].

Futhark Naive Grad

Instead of implementing the rasterizer gradient function manually like in the code supplied by Kerbl et al.[14], I instead wrote a gradient function called **naive_grad** that utilizes the AD function **vjp** in Futhark. To accomplish this, **naive_grad** does not take $\frac{\partial \mathcal{L}}{\partial \mathbf{I}}$, the derivative of the loss with respect to each pixel, as an argument like the reference implementation does. Instead, **naive_grad** calls a loss function defined in Futhark called **loss**. By calculating the loss in Futhark instead of Pytorch, we can assemble a single function, called **forward_naive**, that takes the scene parameters, the camera parameters, and the ground truth image as arguments and returns a loss. Supplying **forward_naive** as an argument to **vjp** then produces the gradient of the loss with respect to the scene parameters. Calculating $\frac{\partial \mathcal{L}}{\partial \theta}$ directly for an arbitrary scene parameter, θ, avoids having to apply the chain rule $\frac{\partial L}{\partial \theta} = \sum_{p \in \mathbf{I}} \frac{\partial L}{\partial p} \frac{\partial p}{\partial \theta}$ where $\mathbf{I}$ is an array of pixels. To reiterate, the reference CUDA implementation takes $\frac{\partial L}{\partial \mathbf{I}}$ as input and performs the aforementioned chain rule computation. By exploiting **vjp**, AD in Futhark avoids manually performing a chain rule step over pixels. In effect, **grad_naive** is able to calculate almost every required gradient with a single vector jacobian product on **forward_naive**, which is highlighted in pink. Please note that the derivative notation ($\frac{\partial \mathcal{L}}{\partial \boldsymbol{\mu}}$, $\frac{\partial \mathcal{L}}{\partial \tilde{\boldsymbol{\mu}}}$, $\frac{\partial \mathcal{L}}{\partial \mathbf{c}}$, or $\frac{\partial \mathcal{L}}{\partial \sigma}$, for example) in the code sample represents variable names, not mathematical operations:

```
-- a version of grad that performs vjp on a unified rasterize/loss function
entry grad_naive [n] (background: [3]f32) (μ: [n][3]f32) (c: [n][3]f32) (σ: [n][1]f32)
                     (s: [n][3]f32) (q: [n][4]f32) (view_matrix: [4][4]f32) (tan_fovx: f32)
                     (tan_fovy: f32) (image_height: i64) (image_width: i64)
                     (ssim_kernel_size: i32) (ssim_kernel_sigma: f32)
                     (gt_image: [image_height][image_width][3]f32) (lambda: f32)
: ([n][3]f32, [n][3]f32, [n][3]f32, [n][1]f32, [n][3]f32, [n][4]f32,
[image_height][image_width][3]f32, [n]i32, f32) =

  -- define a unified function that calculates the loss from the
  -- scene parameters
  let forward_naive =
    \(μ, c, σ, s, q) ->
      (

       -- calulate the radii and pixels
       let (radii, I) = rasterize ... -- params omitted

       -- compute the loss
       let L = loss I ... -- params omitted

       -- return everything
       in (L, radii, I))
```

```
-- inputs to forward_naive
let inps = (μ, c, σ, s, q)

-- Now, we perform a full forward and backward pass on our loss calculation.
-- Note that vjp2 is exactly like vjp except instead of returning a vector
-- jacobian product, it also returns the primal result of the forward primal trace as
-- the first element of a tuple
let ((ℒ, radii, I), (∂ℒ/∂μ, ∂ℒ/∂c, ∂ℒ/∂σ, ∂ℒ/∂s, ∂ℒ/∂q)) =
  vjp2 forward_naive inps (1.0, rep 0, rep (rep [0, 0, 0]))
...
```

Unfortunately, the densification step in the training loop requires $\frac{\partial L}{\partial \hat{\mu}_i}$, which does not correspond to an input to **rasterize**. To calculate $\frac{\partial L}{\partial \hat{\mu}_i}$, therefore, **grad_naive** computes $\hat{\mu}$ independently and defines another function called **forward_naive2** which takes $\hat{\mu}$ instead of μ as input. **forward_naive2** continues from the previous code segment. Again, the call to **vjp** is highlighted in pink:

```
...
let h = i32.i64 image_height
let w = i32.i64 image_width

-- define the same forward pass as last time, but it takes μ̂ instead of μ
let forward_naive2 =
  \μ̂ ->
    (-- calculate the gaussians using algorithm 3
     let gaussians = compute2dGaussians ... -- params omitted

     -- replace the μ̂ we just calculated with the μ̂ given as a parameter to
     -- forward_naive2
     let gaussians' = map2 (\(g: Gaussian2D) (μ̂ᵢ)′ -> g with μ̂ᵢ = (μ̂ᵢ)′) gaussians μ̂

     -- rasterize the gaussians using algorithm 1
     let (_, I) = rasterize2dGaussians gaussians' ... -- params omitted

     -- calculate the loss and return
     in loss .. --params omitted

-- calculate μ̂ from μ.  μ̂ is in clip space here
let μ̂ =
  map (\μᵢ ->
         let cmean = transform_point_4x3 μᵢ (transpose view_matrix)
         in (cmean.0 / (cmean.2 * tan_fovx), cmean.1 / (cmean.2 * tan_fovy)))
      μ

let ∂ℒ/∂μ̂ = vjp forward_naive2 μ̂ 1.0

-- put ∂ℒ/∂μ̂ in the format the pytorch training loop expects
let (∂ℒ/∂μ̂)′ = map (\(x, y) -> [x, y, 0]) ∂ℒ/∂μ̂

in (∂ℒ/∂μ, (∂ℒ/∂μ̂)′, ∂ℒ/∂c, ∂ℒ/∂σ, ∂ℒ/∂s, ∂ℒ/∂q, I, radii, ℒ)
```

The main trick in **forward_naive2** that I use to calculate $\frac{\partial \mathcal{L}}{\partial \hat{\mu}}$ is highlighted in yellow. The idea is that the operations performed in **forward_naive2** are almost the same as in **forward_naive**: Both are just **rasterize2dGaussians** (algorithm 1) composed with **compute2dGaussians** (algorithm 3). The only difference is that **forward_naive2** overwrites the value it calculates for $\hat{\mu}$ with the one given as an argument. This ensures a complete dependency chain between the $\hat{\mu}$ inputted to **forward_naive2** and the loss **forward_naive2** outputs. As a result, **vjp** will propagate adjoints through to the $\hat{\mu}$ inputted as argument to **forward_naive2**, giving us $\frac{\partial \mathcal{L}}{\partial \hat{\mu}}$. The downside to this approach is that **rasterize2dGaussians** (algorithm 1) must be run twice with **vjp**: once in **forward_naive** and again in **forward_naive2**.

Futhark Grad

With **grad**, I sought to solve the issue of double-calling **rasterize2dGaussians** like in **grad_naive**. My technique was to design an inner function called **forward** that takes $\hat{\mu}, \hat{\Sigma}^{-1}, \mathbf{c}, \sigma, \mathbf{s}$, and $\mathbf{q}$ as input. Because μ is not an input to **forward**, $\frac{\partial \mathcal{L}}{\partial \mu}$ is not as easy to calculate. To calculate $\frac{\partial \mathcal{L}}{\partial \mu_i}$, we must use $\frac{\partial \mathcal{L}}{\partial \mu_i} = \frac{\partial \mathcal{L}}{\partial \hat{\mu}_i}\frac{\partial \hat{\mu}_i}{\partial \mu_i} + \frac{\partial \mathcal{L}}{\partial \hat{\Sigma}_i^{-1}}\frac{\partial \hat{\Sigma}_i^{-1}}{\partial \mu_i}$. My implementation of **forward** performs the Gaussian projection but

overwrites the computed values for $\hat{\boldsymbol{\mu}}$ and $\hat{\Sigma}^{-1}$ with the values passed as arguments to the function. This is the same gradient propagation trick that I highlighted in yellow in **grad_naive**, but it means that $\hat{\boldsymbol{\mu}}$ and $\hat{\Sigma}^{-1}$ are calculated twice, thereby duplicating some work to enable ergonomic use of **vjp**. My final **forward** function follows, with the variable replacement trick highlighted in yellow again:

```
entry grad [n] (background: [3]f32) (μ: [n][3]f32) (c: [n][3]f32) (σ: [n][1]f32)
                     (s: [n][3]f32) (q: [n][4]f32) (view_matrix: [4][4]f32) (tan_fovx: f32)
                     (tan_fovy: f32) (image_height: i64) (image_width: i64)
                     (ssim_kernel_size: i32) (ssim_kernel_sigma: f32)
                     (gt_image: [image_height][image_width][3]f32) (lambda: f32)
: ([n][3]f32, [n][3]f32, [n][3]f32, [n][1]f32, [n][3]f32, [n][4]f32,
[image_height][image_width][3]f32, [n]i32, f32) =
  ...
  -- define a forward pass to get the loss
  let forward =
    \(μ̂, Σ̂⁻¹, c, σ, s, q) ->
      (-- calculate the 2d gaussians (algorithm 3$)
       let gaussians =
         compute2dGaussians μ c σ s q view_matrix tan_fovx tan_fovy h w

       -- replace the screen space means and conic values we just calculated with
       -- the ones we were given to ensure correct propagation of gradients with vjp
       let gaussians' = map3 (\(g: Gaussian2D) (μ̂ᵢ)' (Σ̂ᵢ⁻¹)' -> (g with μ̂ᵢ = (μ̂ᵢ)')
                                with Σ̂ᵢ⁻¹ = (Σ̂ᵢ⁻¹)') gaussians μ̂ Σ̂⁻¹

       -- rasterize the gaussians (algorithm 1 in my paper)
       let (radii, I) = rasterize2dGaussians gaussians' ... -- params omitted

       -- calculate the loss
       let ℒ = loss I ... --params omitted

       in (ℒ, radii, I))

  -- inputs to our forward pass
  let inps = (μ̂, Σ̂⁻¹, c, σ, s, q)

-- perform a full forward and backward pass on our loss calculation
  let ((ℒ, radii, I), (∂ℒ/∂μ̂, ∂ℒ/∂Σ̂⁻¹, ∂ℒ/∂c, ∂ℒ/∂σ, _, _)) =
        vjp2 forward inps (1.0, rep 0, rep (rep [0, 0, 0]))
  ...
```

Notice that when **vjp2** is called on the last line (highlighted in pink), the gradients $\frac{\partial \mathcal{L}}{\partial \mathbf{s}}$ and $\frac{\partial \mathcal{L}}{\partial \mathbf{q}}$, corresponding to the scales and rotations, respectively, are computed but ignored. This is because the scales and rotations are used to calculate the value Σ_i^{-1} which is subsequently overwritten, implying that the outputted gradients for the scales and rotations are 0. This is because the scales and rotations passed to **forward** do not contribute to the loss. Practically, this means that $\frac{\partial \mathcal{L}}{\partial \mathbf{s}}$ and $\frac{\partial \mathcal{L}}{\partial \mathbf{q}}$ must be calculated using the chain rule like we did with $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\mu}_i}$. The **grad** function continues on to perform these chain rule operations with:

```
  -- use the chain rule to calculate ∂ℒ/∂μ = ∂ℒ/∂μ̂ ∂μ̂/∂μ + ∂ℒ/∂Σ̂⁻¹ ∂Σ̂⁻¹/∂μ
  let ∂ℒ/∂μ =
    map (\(i: i64) ->
           [ ∂μ̂/∂μ[i][0][0] * ∂ℒ/∂μ̂[i].0 + ∂μ̂/∂μ[i][1][0] * ∂ℒ/∂μ̂[i].1 + ∂Σ̂⁻¹/∂μ[i][0].0 * ∂ℒ/∂Σ̂⁻¹[i].0 +
               ∂Σ̂⁻¹/∂μ[i][0].1 * ∂ℒ/∂Σ̂⁻¹[i].1 + ∂Σ̂⁻¹/∂μ[i][0].2 * ∂ℒ/∂Σ̂⁻¹[i].2
           , ∂μ̂/∂μ[i][0][1] * ∂ℒ/∂μ̂[i].0 + ∂μ̂/∂μ[i][1][1] * ∂ℒ/∂μ̂[i].1 + ∂Σ̂⁻¹/∂μ[i][1].0 * ∂ℒ/∂Σ̂⁻¹[i].0 +
               ∂Σ̂⁻¹/∂μ[i][1].1 * ∂ℒ/∂Σ̂⁻¹[i].1 + ∂Σ̂⁻¹/∂μ[i][1].2 * ∂ℒ/∂Σ̂⁻¹[i].2
           , ∂μ̂/∂μ[i][0][2] * ∂ℒ/∂μ̂[i].0 + ∂μ̂/∂μ[i][1][2] * ∂ℒ/∂μ̂[i].1 + ∂Σ̂⁻¹/∂μ[i][2].0 * ∂ℒ/∂Σ̂⁻¹[i].0 +
                ∂Σ̂⁻¹/∂μ[i][2].1 * ∂ℒ/∂Σ̂⁻¹[i].1 + ∂Σ̂⁻¹/∂μ[i][2].2 * ∂ℒ/∂Σ̂⁻¹[i].2
           ])
         (iota n)

  -- listify ∂ℒ/∂μ̂ for backward compatibility
  let (∂ℒ/∂μ̂)' = map (\m -> [m.0, m.1, 0]) ∂ℒ/∂μ̂
```

```
-- calculate ∂L/∂s = ∂L/∂Σ̂⁻¹ ∂Σ̂⁻¹/∂s
let ∂L/∂s = map2 transform_tup_3x3 ∂L/∂Σ̂⁻¹ ∂Σ̂⁻¹/∂s
-- calculate ∂L/∂q = ∂L/∂Σ̂⁻¹ ∂Σ̂⁻¹/∂q
let ∂L/∂q = map2 transform_tup_4x3 ∂L/∂Σ̂⁻¹ ∂Σ̂⁻¹/∂q
in (∂L/∂μ, (∂L/∂μ̂)′, ∂L/∂c, ∂L/∂σ, ∂L/∂s, ∂L/∂q, I, radii, L)
```

where the functions **transform_tup_3x3** and **transform_tup_4x3** are utility functions for matrix multiplication, and $\frac{\partial \hat{\Sigma}^{-1}}{\partial \mathbf{q}}$, $\frac{\partial \hat{\Sigma}^{-1}}{\partial \mathbf{s}}$, $\frac{\partial \hat{\Sigma}^{-1}}{\partial \boldsymbol{\mu}}$, and $\frac{\partial \hat{\boldsymbol{\mu}}}{\partial \boldsymbol{\mu}}$ are Jacobian matrices calculated with Futhark AD earlier in **grad**. The first part of **grad** used to compute these Jacobians is not shown in this section.

In summary, the need to calculate $\frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{\mu}}}$ complicates **grad** substantially. We are forced to depart from the AD convention of relying solely on the compiler to handle gradients for us, and instead must analyze the dependencies present within **rasterize** to employ the chain rule manually. However, even after constructing **grad** to output $\frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{\mu}}}$, our code is still able to leverage AD and avoid a large amount of manual derivation present in the reference CUDA implementation[14].

Pixel Color Loop Cutoff

When profiled, I found that the majority of the time spent running **grad** was spent calculating the derivative of the pixel color function (see figure 3.3). Originally, I had crafted a faithful implementation of algorithm 1 called **pixel_color_test**, but after careful analysis of the IR code produced by the Futhark compiler, I realized that the while loop I had used to iterate over each Gaussian in each pixel's tile had introduced significant overhead when coupled with **vjp**. The issue was that the compiler was unable to determine in advance how many iterations the while loop would run for, and it therefore could not easily compute how much memory to allocate for the tape. To perform AD on the while loop in **pixel_color_test**, the compiler had written code to run the entire loop once (the same loop as from algorithm 1) solely to count loop iterations before allocating the tape. The entire loop would then run again, this time using the allocated tape to record the values of intermediate variables across iterations. Obviously, it is unacceptable to perform the pixel color calculation—one of the expensive steps in the algorithm—any more than absolutely necessary.

Using **pixel_color_test** to train a scene was ruinously slow in practice as well as in theory. I therefore chose to write a different version of the pixel color calculation using a for loop instead. I kept both functions in my 3DGS implementation and named the for loop version **pixel_color_train**. During normal operation, **pixel_color_test** is used by **rasterize** for rendering because its performance is less catastrophic when it is called on its own instead of inside **vjp**. On the other hand, **pixel_color_train** is reserved for training contexts because it only approximates algorithm 1, trading physical accuracy for speed by avoiding the expensive precompute step when used in conjunction with **vjp**. The for loop in **pixel_color_train** will always loop for an empirically determined 2^{11} iterations per invocation. This design decision greatly hastened the training process in general, but the advantage became especially obvious after a few dozen rounds of densification had increased the total number of Gaussians in the scene.

3.2.3 Code Integration with PyTorch Training Loop

My strategy for integrating the Futhark rasterizer with the existing PyTorch code was to mimic the framework used by the original authors to integrate their CUDA rasterizer. I copied the **_RasterizeGaussians** class and modified it to wrap my compiled Futhark binary instead of the reference CUDA code. I then added an additional **--futhark** flag to the arguments of the training loop to toggle the use of my Futhark rasterizer. To call Futhark functions from Python, I used a forked version of the futhark-server-python project[4] that I had modified to allow the version of Python used by Kerbl et al. (Python 3.7). To wrap the Futhark **_RasterizeGaussian** class, I also modified a copy of the **render** function, which I renamed **futhark_render**. Within the training loop, I modified every invocation of **render** to ensure that the correct render function was called based on the command-line **--futhark** flag.

3.2.4 Evaluation

To test my implementation, I used the four scenes available for download on the 3DGS paper website[14]: Train, Truck, Dr. Johnson, and Playroom. Train and Truck are scenes originally from the Tanks and Temples dataset[16], whereas Dr. Johnson and Playroom originate from the Deep Blending dataset[7]. I chose these scenes for the same reason as the 3DGS authors likely did: They offer a good balance of indoor and outdoor environments with different lighting, textures, and total area. To ensure consistency between runs, I employed the test/train split used by 3DGS[14] and NeRF[19], withholding every 8th photo for the test set.

Following in the footsteps of the 3DGS authors[14], I chose to only train each scene once. My logic was that the individual iteration times would average across 30k iterations of training, implying that distinct training runs would not differ meaningfully in wall-clock duration. Furthermore, following Kerbl et al.[14], I assumed that the variance in the final scene quality between training runs would not be significant.

All training was conducted on a cluster node running an NVIDIA Titan RTX with the command

```
python train.py \
   --convert_SHs_python \
   --iterations 30000 \
   --checkpoint_iterations 1 10000 20000 30000\
   --test_iterations 10 1000 5000 10000 15000 20000 25000 30000 \
   --save_iterations 30000 \
   -s "$DATASET" \
   --eval \
   --disable_viewer
   --futhark
```

However, I omitted the **`futhark`** argument when testing the reference CUDA implementation. Training was run inside the conda environment loaded from the **`environment.yml`** file provided by the 3DGS authors in their repository.

The performance metrics PSNR, SSIM[28], and LPIPS[30] were calculated using the reference scripts **`render.py`** and **`metrics.py`** as described by the 3DGS authors in their **`README.md`**. Additionally, I modified the training loop (**`train.py`**) to allow a **`--futhark_profile <N>`** flag, which takes an integer argument and can be used to generate a .json profiling file. This file can, in turn, be given as an argument to **`futhark profile`** to produce a human-readable profiling result of my Futhark implementation's runtime performance. **`--futhark_profile <N>`** instructs the training loop to pass the runtime profiling flag to the Futhark server upon its construction. The server will then gather profiling information over however many iterations are specified by the flag. To gather profiling information without interfering with normal training operation, I periodically checkpointed the training run and, after training had completed, profiled the Futhark rasterizer for 200 iterations from each saved checkpoint.

To test the rendering speed of the Futhark and CUDA code, I devised a custom test script. The script first renders 10 warm-up frames before timing a single full orbit of the scene over the next 100 frames. During the orbit, the camera remains approximately 5 units away from the world-space origin, which it stays pointed at throughout. Each frame is rendered at a resolution of 980×545 with a horizontal field of view of 1.4 radians and a vertical field of view of 0.9 radians. All frames rendered by Futhark in the rendering speed test used the **`rasterize`** function.

3.3 Results

I found that the Futhark rasterizer was often over 10 times slower than the reference implementation in training and over 30 times slower during rendering (see figure 3.1). Furthermore, due to the pixel color loop cutoff discussed in section 3.2.2, the convergence of the Futhark implementation also suffers (see figs 3.4 and 3.2).

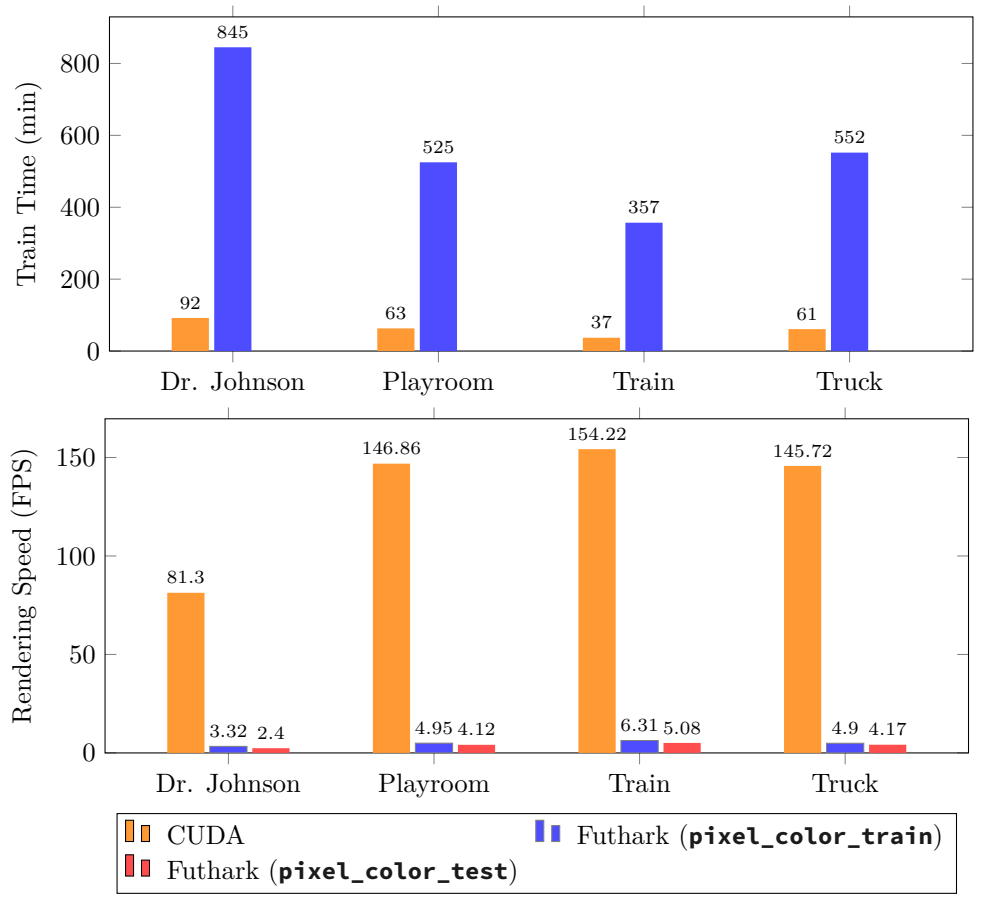

Figure 3.1: **Performance.** On the top, I track how many minutes of wall-clock time 30k training iterations take for the Futhark and CUDA implementations respectively. The time includes saving 4 checkpoints but not the initial time taken to load the scene. On the bottom, I calculate the throughput of each renderer while rendering a scene trained for 30k iterations. The red bar represents the performance of the renderer during standard rendering operation where **`pixel_color_test`** is used, and there is no upper limit on the number of Gaussians that can contribute to the calculation of any single pixel in the frame. The blue bar represents a modified version of rendering using **`pixel_color_train`**, capping the number of Gaussians evaluated at each pixel to a maximum of 2^{11}. The idea behind the capped metric is to facilitate a comparison, between the two plots, of the relative performance of the Futhark and CUDA implementations.

During each throughput test, the camera orbits around the center of the scene to ensure that all parts of the scene are exposed to the camera somewhat evenly. FPS is then calculated over 100 frames, excluding 10 warm-up frames. All rendering frames were rendered at 980×545 pixels

Figure 3.2: **Qualitative Comparison**: A comparison between the Futhark and CUDA renderer outputs after 30k training iterations. The left-hand column of images is from the *train* dataset, whereas the right-hand images are from the *truck* dataset. Futhark renders look a little fuzzy around the edges of some objects because of the use of the for loop in the pixel color calculation. All images are from the testing set, meaning that the scene was not trained on them.

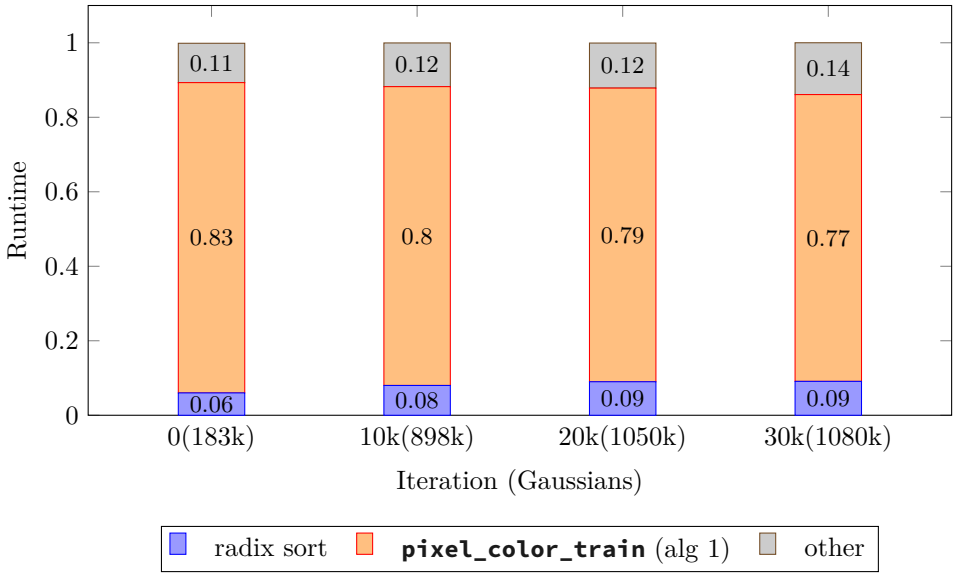

Figure 3.3: **Profiling results:** The fraction of runtime spent on each function of my Futhark rasterizer while training the "Train" scene from Tanks and Temples[16]. In this graphic, **`pixel_color_train`** corresponds with the **`pixel_color_train`** function described in section 3.2.2. These numbers were generated by resuming training from a checkpoint with the command line argument **`--futhark_profile 200`** given to the **`train.py`** file. The performance of the Futhark rasterizer was then profiled for the next 200 training iterations, and the results were processed with **`futhark profile`**.

Scene	Renderer	PSNR ↑	SSIM ↑	LPIPS ↓	Gaussians
Dr. Johnson	CUDA	29.36	0.903	0.239	3,074,790
	Futhark	25.39	0.856	0.293	3,213,537
Playroom	CUDA	30.06	0.903	0.243	1,842,485
	Futhark	27.59	0.880	0.272	2,009,642
Train	CUDA	22.12	0.818	0.198	1,080,004
	Futhark	20.99	0.79	0.226	1,047,913
Truck	CUDA	25.46	0.881	0.143	2,060,941
	Futhark	23.61	0.852	0.168	2,076,380

Figure 3.4: **Quantitative Comparison**. I compare the CUDA and Futhark rasterizer backends across the four scenes from the Tanks and Temples[16] and Deep Blending[7] datasets. PSNR, SSIM[28], and LPIPS[30] are computed on held-out test views at the final training iteration (30,000). The "Gaussians" column gives the number of Gaussians used to represent the scene by the time training completes.

3.4 Discussion

3.4.1 Performance and Quality

The Futhark implementation of 3DGS was outperformed by the reference implementation—unsurprising on its own, given that AD is almost always slower than equivalent manual gradient functions. What is remarkable is the extent of that gap, which should warrant further examination of why Futhark's AD struggles for this class of problem. As seen in figure 3.3, the majority of training time is spent on **`pixel_color_train`**. I am fairly certain that a reason for this severe performance mismatch is due to the reliance of reverse mode AD on the tape to recover intermediate variables saved during the primal trace. This design characteristic of AD necessarily converts the gradient function into a memory bound computation. On the other hand, the reference implementation can apply an inverse for every variable in the primal and thereby recover intermediate variables using only the registers (see algorithms 5 and 1). In a massively-parallel context, like the Titan RTX used my training, reducing expensive memory lookups can yield massive performance improvements.

Another reason for the large performance difference between the Futhark and CUDA implementations is the incredible skill with which the CUDA version was designed. In particular, the 3DGS authors used a clever scheme for loading each tile's Gaussians into shared memory before the pixel color loop. Their realization was that they could take advantage of how every pixel in a tile uses the same set of Gaussians in the pixel color calculation. To make use of this pattern, they assign one thread block per tile, and one thread per pixel within the tile. Each thread then loads one of the next 256 Gaussians into shared memory and syncs with the other threads before continuing. Lastly, every thread then loops through the next 256 Gaussians in shared memory to complete the next chunk of the pixel color calculation. This process then repeats until every thread has calculated the color of its pixel. In effect, this technique facilitates a strided memory access pattern and allows threads that have finished calculating the color of their assigned pixel a chance to contribute to the color calculation of other pixels by loading Gaussians into shared memory. They are able to employ this collaborative loading technique in both the primal and manual gradient pixel color functions.

Futhark, on the other hand, does not allow for these types of low-level optimizations because the entire language is designed to only express operations that carry semantic effect. Without access to tricks like the ones used by the 3DGS authors, Futhark's comparatively poor performance becomes much easier to explain. This theory matches the results in figure 3.3 showing that the vast majority of the runtime of my 3DGS implementation is spent on the pixel color for loop, **`pixel_color_train`**, which corresponds to the most heavily optimized code section in the CUDA reference. This result was expected and adds evidence to the validity of my Futhark implementation. If my code were poorly written, it would be possible for another category to dominate the runtime or for the profiling results to vary wildly throughout training.

Interestingly, the relative performance of the Futhark implementation is much worse during rendering than during training. Averaged across the four scenes, the Futhark implementation is around 9× slower during training but around 27× slower during rendering, which is counterintuitive given our understanding that AD is slower than manual gradient calculation. This result is not explained by variations in the Gaussian count during training[3], nor by the different pixel color methods used by **`train`** and **`rasterize`** (the 27× slower figure uses the **`pixel_color_train`** function in both training and rendering).

The cause likely lies in the reference CUDA implementation rather than mine. In the CUDA version, the manual gradient function accumulates per-Gaussian gradients using atomic adds, with every pixel in a tile contending for write access to the same set gradients. Unlike the memory bandwidth bottleneck in the forward pass, which the CUDA reference mitigates by cooperatively loading Gaussians into shared memory, there is no effective mitigation for the contention on write locks while calculating gradients for each gaussian. Even though the manual gradient calculation in CUDA is faster than the automatic differentiation in Futhark, structural limitations may prohibit the reference CUDA implementation from outperforming Futhark in the same way as during rendering. To confirm this theory, a careful profiling would have to be performed on the render and gradient CUDA kernels, measuring whether the backward pass is throttled by atomic operations in a way that the forward pass is not.

My use of the for loop in **`pixel_color_train`** allowed me to practically assess the training capabilities of my Futhark implementation, but it also imposed a marked decrease in image quality of the trained scene. This quality dip is shown in the metrics given in figure 3.4 but is also visible in the qualitative example provided in figure 3.2. In particular, we see that the borders of solid objects are sometimes "fuzzy", presumably from under-optimized Gaussians gathering on their surfaces.

3.4.2 The Futhark Advantage

In exchange for the performance drop that comes with AD, the programmer is relieved of having to calculate gradients manually. In an ideal setting, such a trade-off would allow the programmer to calculate every gradient with a "one-liner" consisting of a single call to **`vjp`**. Unfortunately, the reliance of 3DGS on the screen-space mean gradient, $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\mu}}$, complicates **`grad`**. We are forced to decompose the primal, **`rasterize`**, into its constituent functions and use AD to calculate their derivatives individually before computing our target derivatives with the chain rule. This process, while easier to implement than a manual gradient function in CUDA, is far from ideal. At the same time, it is not immediately obvious how a programming language should support the derivation of intermediate results like $\boldsymbol{\mu}$ in **`rasterize`**. Futhark's **`vjp`** and **`jvp`** specify which variables to differentiate with respect to via the parameters of the lambda passed to them. However, this mechanism breaks down when differentiating with respect to intermediate variables which are hidden within the scope of the lambda and are therefore inaccessible to the higher-order **`vjp`** or **`jvp`**.

Despite the complexity of computing $\frac{\partial \mathcal{L}}{\partial \tilde{\boldsymbol{\mu}}}$ in **`grad`**, programming in Futhark is substantially easier and arguably more enjoyable than low-level CUDA programming. My finished implementation consists of far fewer files, lines of code, and functions than the CUDA implementation it sought to replace. As such, my Futhark implementation may have been better suited as a prototyping step in the creation of higher-performance CUDA code. CPU programs are often used as an intermediate step in the development of GPU programs, but this development pipeline would be quite difficult to achieve for large projects like 3DGS due to the comparatively slow speed of the CPU in data-parallel contexts. Despite running much slower than its CUDA counterpart, my Futhark implementation would almost certainly outperform a CPU 3DGS program by a large margin, making it an attractive option for testing a proof of concept. Futhark, therefore, provides a welcome addition to the development pipeline by allowing large GPU programs to be easily tested without having to debug endless CUDA errors or spend days waiting for a CPU-computed result.

A major advantage of using AD is the ability to modify the primal function with only minimal changes to its gradient. In the context of this thesis, we could modify my primal **`rasterize`** function to improve 3DGS with only minimal changes to **`grad`**. A natural extension of my rasterizer would incorporate work by Qi, Cai, and Han[23], that eliminates the error introduced by the local linear approximation of Gaussian projection. In their work, they parameterize the camera-space Gaussians with ellipses and map those to the projection plane instead of the Gaussians directly. To be precise, they project the ellipse parametrized by a three-standard-deviation level set of each Gaussian. This method offers promise because, unlike 3D Gaussians, ellipses are closed under non-affine perspective projection operations. Unfortunately, their approach still suffers from some projection error because the

[3]The time taken per iteration stayed fairly consistent throughout most of the training for both implementations.

centers of the projected ellipses do not exactly match the projected centers of the corresponding camera space ellipses. Nonetheless, Qi, Cai, and Han showed that their method yielded slightly better results than normal 3DGS. Using my Futhark rasterizer, it should be relatively straightforward to update the **`rasterize`** function to eliminate its dependence on the local linear approximation of the projection operator with the technique proposed by Qi, Cai, and Han. Furthermore, only minimal changes would be needed in **`grad`** instead of the large revision that would be required of a manual gradient function.

3.4.3 Potential Improvements to Futhark's AD

One of the original goals of this project was to motivate development of the Futhark compiler, and in that sense, I provide a meaningful contribution. I have created a working implementation of a differentiable renderer that exposes several opportunities for development in Futhark. For example, Futhark could offer the option to add compiler directives for user-supplied inverse functions to reduce reliance on the tape and improve the performance of **`vjp`**. The idea is that the compiler could store the final state of the loop variables along with user-supplied inverse functions for each variable. Then, to calculate the value of a variable one iteration prior in the loop, its corresponding inverse could be applied without having to access memory. This framework is how the manual gradient technique (algorithm 5) employed by Kerbl et al. functions. By reducing reliance on the tape, **`vjp`** could speed up significantly, potentially eliminating the need to use the for loop in **`pixel_color_train`**. Replacing **`pixel_color_train`**, with the while-loop-powered **`pixel_color_test`** would make the semantic meaning of my Futhark implementation identical to the CUDA reference, and, in turn, eliminate the quality drop-off in the Futhark output. In summary, Futhark could abstract standard techniques used in manual differentiation and improve the runtime performance of **`vjp`** by allowing user-supplied inverse functions for loop variables. In turn, these improvements could make an improved version of my Futhark implementation more practical.

3.5 Conclusion

In this thesis, I reviewed the theory behind light propagation, volume rendering, 3D Gaussian Splatting, and automatic differentiation, and combined these concepts into a working Futhark implementation that replaces the manual gradient function in the original 3DGS code with an AD pipeline.

After testing on the datasets provided by the 3DGS authors, I found that the Futhark 3DGS renderer described in this paper is many times slower than the reference CUDA implementation, both in training and rendering. This gap is best explained as a combination of two factors: the reliance of reverse mode AD on the tape to recover intermediate values during the adjoint trace, and the absence of low-level CUDA memory tricks that are inaccessible to Futhark. Despite this performance gap, my Futhark implementation is dramatically smaller and simpler than its CUDA counterpart and requires much less explicit derivative math. This suggests that Futhark is better suited as a stage in the development of GPU programs rather than a final result.

The ability of AD in Futhark to accommodate significant changes to a primal function with minimal modification to its associated gradient function is one of the things that will make it shine in the prototyping role. Ease of use in this setting can be more useful than raw performance, because it allows the programmer to more freely experiment with new ideas. For example, I hypothesize that it would be much easier to implement the technique proposed by Qi, Cai, and Han[23] in the Futhark version than in the original CUDA.

With any luck, this thesis will contribute to the further development of Futhark, potentially highlighting the usefulness of user-supplied inverse functions for computationally expensive sections of the primal. Ultimately, Futhark's value is not as a finished renderer but as a testbed for ideas that would otherwise be too costly to implement in a lower level language. If this thesis nudges Futhark's development toward greater performance without sacrificing core usability in that context, it will have succeeded.

Chapter 4

Appendix

4.1 Solving the Emission-Absorption Differential Equation

Recall the emission-absorption equation 2.2

$$\frac{dC_\lambda(s)}{ds} = \tau(s)C_\lambda(s) - g(s).$$

which governs the rate of change of radiance at wavelength λ with respect to distance s along the viewing ray from the observer. Following the derivation technique introduced by Max[18], we start by moving the attenuation term $\tau(s)C_\lambda(s)$ to the left side and multiplying both sides by the integrating factor $e^{\int_s^D \tau(t)dt}$:

$$\frac{dC_\lambda(s)}{ds}e^{\int_s^D \tau(t)dt} - \tau(s)C_\lambda(s)e^{\int_s^D \tau(t)dt} = -g(s)e^{\int_s^D \tau(t)dt}. \tag{4.1}$$

Now, we can apply the inverse product rule and obtain[1]

$$\frac{d}{ds}\left(C_\lambda(s)e^{\int_s^D \tau(t)dt}\right) = -g(s)e^{\int_s^D \tau(t)dt}. \tag{4.2}$$

We can now integrate both sides from the observer, $s = 0$, to the arbitrary distance, $s = D$, with

$$\begin{aligned}
\int_0^D \frac{d}{ds}\left(C_\lambda(s)e^{\int_s^D \tau(t)dt}\right)ds &= \int_0^D -g(s)e^{\int_s^D \tau(t)dt}ds \qquad (4.3)\\
C_\lambda(D)e^{\int_D^D \tau(t)dt} - C_\lambda(0)e^{\int_0^D \tau(t)dt} &= \int_0^D -g(s)e^{\int_s^D \tau(t)dt}ds\\
-C_\lambda(0)e^{\int_0^D \tau(t)dt} &= -C_\lambda(D) - \int_0^D g(s)e^{\int_s^D \tau(t)dt}ds\\
C_\lambda(0)e^{\int_0^D \tau(t)dt} &= C_\lambda(D) + \int_0^D g(s)e^{\int_s^D \tau(t)dt}ds.
\end{aligned}$$

where we used the Fundamental Theorem of Calculus[2] to apply the integral on the left side of equation 4.3. We interpret $C_\lambda(D)$ to mean the radiance of light along the viewing ray at distance D from the observer, and we similarly interpret $C_\lambda(0)$ as the radiance of light reaching the observer. Next, we

[1] Integrating factors are common in solutions to differential equations but can appear strange to those unfamiliar with the field. To understand the purpose of the integrating factor, note how if we were to apply the derivative on the left side of equation 4.2 using the product rule, we would be left with equation 4.1. In essence, the integrating factor allows us to go in the opposite direction using the inverse product rule to consolidate the summed terms (in equation 4.1) into a product (in equation 4.2). If attempting to apply the product rule on equation 4.2, notice that $\frac{d}{ds}e^{\int_s^D \tau(t)dt} = \frac{d}{ds}e^{-\int_D^s \tau(t)dt} = -\tau(s)e^{-\int_D^s \tau(t)dt} = -\tau(s)e^{\int_s^D \tau(t)dt}$.

[2] As a reminder, the second part of the Fundamental Theorem of Calculus states that $\int_a^b \frac{d}{dx}F(x)dx = F(b) - F(a)$

multiply both sides by $e^{-\int_0^D \tau(t)dt}$ to solve for $C_\lambda(0)$:

$$\begin{aligned}
C_\lambda(0) &= e^{-\int_0^D \tau(t)dt} C_\lambda(D) + e^{-\int_0^D \tau(t)dt} \int_0^D g(s) e^{\int_s^D \tau(t)dt} ds \\
&= e^{-\int_0^D \tau(t)dt} C_\lambda(D) + \int_0^D g(s) \left(e^{\int_s^D \tau(t)dt} \right) \left(e^{-\int_0^D \tau(t)dt} \right) ds \\
&= e^{-\int_0^D \tau(t)dt} C_\lambda(D) + \int_0^D g(s) \left(e^{\int_s^D \tau(t)dt - \int_0^D \tau(t)dt} \right) ds \\
&= e^{-\int_0^D \tau(t)dt} C_\lambda(D) + \int_0^D g(s) \left(e^{\int_s^D \tau(t)dt - \left(\int_0^s \tau(t)dt + \int_s^D \tau(t)dt \right)} \right) ds \\
&= \underbrace{e^{-\int_0^D \tau(t)dt} C_\lambda(D)}_{\substack{\text{radiance contributed} \\ \text{from } \textbf{outside} \text{ medium}}} + \underbrace{\int_0^D g(s) e^{-\int_0^s \tau(t)dt} ds}_{\substack{\text{radiance contributed} \\ \text{from } \textbf{inside} \text{ medium}}}
\end{aligned} \tag{4.4}$$

where the first term of equation 4.4 represents the radiance entering the medium at distance $s = D$ attenuated by absorption along the ray. The integral term describes the total light emitted within the medium along the viewing ray.

The source term is often written as $g(s) = c_\lambda(s)\tau(s)$ where the *emission coefficient* c_λ corresponds to the radiance of wavelength λ contributed per unit of extinction. Furthermore, it is often assumed that $C_\lambda(D) = 0$, and that all of the light along the ray was emitted by the medium. Lastly, we write $C_\lambda(0)$ as C_λ to signify the radiance at wavelength λ at the "viewing end" of the ray. We assume that if this ray were to hit an observer, then that observer would be at position $s = 0$ on the ray and therefore would perceive light at wavelength λ with radiance C_λ. After applying these conventions, we are left with our final equation:

$$C_\lambda = \int_0^D c_\lambda(s)\tau(s)e^{-\int_0^s \tau(t)dt} ds.$$

We use this result in section 2.2

4.2 Solving the Emission-Absorption Approximation

Our goal is to find a numerical approximation for equation 2.3. Our final result will match the one used in NeRF[19], and our approach will follow Tagliasacchi's[27]. We start by splitting the outer integral into a summation of n separate integrals representing consecutive, nonoverlapping ray segments[27]. We now have

$$C_\lambda = \sum_{i=1}^{n} \int_{s_i}^{s_{i+1}} c_\lambda(s)\tau(s)T(0 \to s) ds$$

where $T(0 \to s) = T(s) = e^{-\int_0^s \tau(t)dt}$ expresses the transmittance between the points 0 and s on the ray, and s_i is the start of the ith ray segment. Because s always adopts a value between s_i and s_{i+1}, we can introduce the approximation $s \approx s_i$[27]. Now, we have

$$\begin{aligned}
C_\lambda &\approx \sum_{i=1}^{n} \int_{s_i}^{s_{i+1}} c_\lambda(s_i)\tau(s_i)T(0 \to s) ds \\
&= \sum_{i=1}^{n} \int_{s_i}^{s_{i+1}} c_\lambda(s_i)\tau(s_i)T(0 \to s_i)T(s_i \to s) ds \\
&= \sum_{i=1}^{n} T(0 \to s_i)c_\lambda(s_i)\tau(s_i) \int_{s_i}^{s_{i+1}} T(s_i \to s) ds.
\end{aligned} \tag{4.5}$$

We turn our attention to solving the inner transmittance integral in equation 4.5. We can introduce another approximation, $\tau(t) \approx \tau(s_i)$, to get

$$\begin{aligned}
\int_{s_i}^{s_{i+1}} T(s_i \to s)ds &= \int_{s_i}^{s_{i+1}} e^{-\int_{s_i}^{s} \tau(t)dt} ds \\
&\approx \int_{s_i}^{s_{i+1}} e^{-\int_{s_i}^{s} \tau(s_i)dt} ds \\
&= \int_{s_i}^{s_{i+1}} e^{-\tau(s_i)(s-s_i)} ds \\
&= \left. \frac{e^{-\tau(s_i)(s-s_i)}}{-\tau(s_i)} \right|_{s=s_i}^{s_{i+1}} \\
&= \left(\frac{e^{-\tau(s_i)\delta_i}}{-\tau(s_i)} \right) - \left(\frac{e^{0}}{-\tau(s_i)} \right) \\
&= \frac{1 - e^{-\tau(s_i)\delta_i}}{\tau(s_i)}
\end{aligned} \tag{4.6}$$

where $\delta_i = s_{i+1} - s_i$ is the length of the ith ray segment in the approximation. Now, we plug equation 4.6 into equation 4.5:

$$\begin{aligned}
C_\lambda &\approx \sum_{i=1}^{n} T(0 \to s_i) c_\lambda(s_i) \tau(s_i) \int_{s_i}^{s_{i+1}} T(s_i \to s)ds && \text{equation 4.5 definition} \\
&\approx \sum_{i=1}^{n} T(0 \to s_i) c_\lambda(s_i) \tau(s_i) \frac{1 - e^{-\tau(s_i)\delta_i}}{\tau(s_i)} && \text{equation 4.6 substitution} \\
&= \sum_{i=1}^{n} T(0 \to s_i) c_\lambda(s_i) \left(1 - e^{-\tau(s_i)\delta_i}\right).
\end{aligned} \tag{4.7}$$

We can approximate $T(0 \to s_i)$ with the Riemann sum

$$\begin{aligned}
T(0 \to s_i) &= e^{-\int_0^{s_i} \tau(t)dt} \\
&\approx e^{-\sum_{j=1}^{i-1} \tau(s_j)\delta_j} \\
&= \prod_{j=1}^{i-1} e^{-\tau(s_j)\delta_j}
\end{aligned}$$

which we plug into equation 4.7 to get our final approximation[27, 19]:

$$C_\lambda \approx \sum_{i=1}^{n} c_\lambda(s_i) \left(1 - e^{-\tau(s_i)\delta_i}\right) \left(\prod_{j=1}^{i-1} e^{-\tau(s_j)\delta_j} \right)$$

We use this result in section 2.2.

4.3 Tristimulus Theory

In practice, the emission coefficient $c_{i,\lambda}$ used in equation 2.12 is entirely unknowable. The issue is that real-world cameras cannot capture radiance at every wavelength for every pixel. Such a measurement would constitute knowledge of the entire spectrum shown to the camera. Instead, cameras detect and store three *tristimulus values* known as C_r, C_g, and C_b defined as

$$\begin{aligned}
C_r(\mathbf{p}) &= \int_\lambda m_r(\lambda) C_\lambda(\mathbf{p}) d\lambda \\
C_g(\mathbf{p}) &= \int_\lambda m_g(\lambda) C_\lambda(\mathbf{p}) d\lambda \\
C_b(\mathbf{p}) &= \int_\lambda m_b(\lambda) C_\lambda(\mathbf{p}) d\lambda
\end{aligned}$$

where m_r, m_g, and m_b are spectral sensitivity functions that account for the sensitivity of the camera's sensor at wavelength λ. These tristimulus values are single scalars describing the entire spectral radiance curve at each pixel weighted by how sensitive each sensor channel is to that wavelength. For example, instead of measuring the exact radiance at 650 nanometers, $C_r(\mathbf{p})$ measures how strongly the camera's red-sensitive sensor responds at $\mathbf{p}$ when exposed to the full spectrum of $\int_\lambda C_\lambda(\mathbf{p})d\lambda$ [11][22, chapter 4.6].

Not only are the ground truth images used to train 3DGS recorded as tristimulus values, but all digital computer screens at the time of publication use tristimulus output. Rather than converting the tristimulus camera input into spectral data, training the model, and converting the resulting spectral output back to tristimulus values, we instead opt to train on the tristimulus data directly. This approach requires us to substitute the formula for C_λ (equation 2.12) into the tristimulus formula for C_k where $k \in \{r, g, b\}$. This substitution yields

$$\begin{aligned}
C_k(\mathbf{p}) &= \int_\lambda m_k(\lambda) C_\lambda(\mathbf{p}) d\lambda \\
&\approx \int_\lambda m_k(\lambda) \left[\sum_{i=1}^{n} c_{i,\lambda} \sigma_i \hat{g}_i(\mathbf{p}) \left(\prod_{j=1}^{i-1} (1 - \sigma_j \hat{g}_j) \right) \right] d\lambda \\
&= \int_\lambda \sum_{i=1}^{n} m_k(\lambda) c_{i,\lambda} \sigma_i \hat{g}_i(\mathbf{p}) \left(\prod_{j=1}^{i-1} (1 - \sigma_j \hat{g}_j) \right) d\lambda \\
&= \sum_{i=1}^{n} \int_\lambda m_k(\lambda) c_{i,\lambda} \sigma_i \hat{g}_i(\mathbf{p}) \left(\prod_{j=1}^{i-1} (1 - \sigma_j \hat{g}_j) \right) d\lambda \\
&= \sum_{i=1}^{n} \left[\int_\lambda m_k(\lambda) c_{i,\lambda} d\lambda \right] \sigma_i \hat{g}_i(\mathbf{p}) \left(\prod_{j=1}^{i-1} (1 - \sigma_j \hat{g}_j) \right) \\
&= \sum_{i=1}^{n} c_{i,k} \sigma_i \hat{g}_i(\mathbf{p}) \left(\prod_{j=1}^{i-1} (1 - \sigma_j \hat{g}_j) \right), k \in \{r, g, b\}
\end{aligned}$$

where $c_{i,k} = \int_\lambda m_k(\lambda) c_{i,\lambda} d\lambda$ is the per-Gaussian tristimulus value. In essence, $c_{i,k}$ describes the spectral emission of the ith Gaussian against the sensitivity function m_k—exactly analogous to the process first used to capture the ground truth image set used for training.

4.4 Manually Deriving the 3DGS Pixel Color Gradient

We examine the approach employed by Kerbl et al. to calculate the derivatives of the inputs to algorithm 1 with respect to a loss function: $\frac{\partial \mathcal{L}_\mathbf{p}}{\partial \mathbf{c}_i(\mathbf{v})}$, $\frac{\partial \mathcal{L}_\mathbf{p}}{\partial \alpha_i}$, $\frac{\partial \mathcal{L}_\mathbf{p}}{\partial \sigma_i}$, $\frac{\partial \mathcal{L}_\mathbf{p}}{\partial \hat{\Sigma}_i^{-1}}$, and $\frac{\partial \mathcal{L}_\mathbf{p}}{\partial \boldsymbol{\mu}_i}$. We will begin by assuming an unoptimized 3DGS model where all n Gaussians are evaluated at every pixel. We will then adapt this model into pseudocode for a usable gradient algorithm (see algorithm 5).

Our general approach will be modeled after reverse mode AD: We will propagate an input adjoint, $\frac{\partial \mathcal{L}_\mathbf{p}}{\partial C}$, through the primal function in reverse until we have established derivatives for the primal inputs. We begin by constructing the primal trace dependency tree (figure 4.1) using variables as nodes for clarity. Figure 4.1 will allow us to propagate $\frac{\partial \mathcal{L}_\mathbf{p}}{\partial C}$ through intermediate variables using the chain rule.

Because algorithm 1 only calculates the color of a single pixel $\mathbf{p}$, its gradient only provides the contribution of that pixel to the gradient of the loss. For any parameter θ_i, we can formalize this concept with

$$\frac{\partial \mathcal{L}}{\partial \theta_i} = \sum_{\mathbf{p} \in P} \frac{\partial \mathcal{L}_\mathbf{p}}{\partial \theta_i}.$$

Intuitively, the total sensitivity of the loss to the Gaussians is obtained through accumulating the contributions of every pixel.

4.4.1 Calculating $\frac{\partial \mathcal{L}_\mathbf{p}}{\partial \mathbf{c}_i(\mathbf{v})}$

We consult figure 4.1 and see that $\mathbf{c}_i(\mathbf{v})$ only influences $\mathcal{L}$ through C. Using the chain rule, we calculate

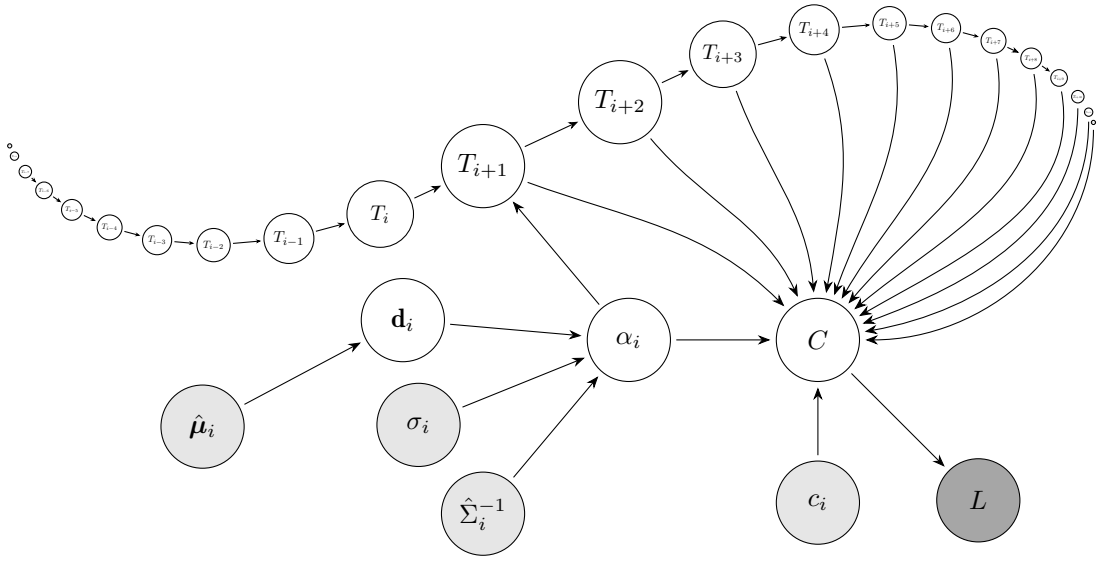

Figure 4.1: **Dependency Tree for Algorithm 1**. Light gray nodes are inputs, dark gray nodes are outputs, and white nodes are helper variables. A parent-child relationship indicates programmatic assignment to the child node variable using the parent node variable in algorithm 1. The chain of T_j nodes indicates a cross-iteration dependency on T. Not all nodes and edges are shown.

$$\frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial \mathbf{c}_i(\mathbf{v})} = \frac{\partial C}{\partial \mathbf{c}_i(\mathbf{v})} \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial C} = \alpha_i T_i \cdot \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial C}. \tag{4.8}$$

4.4.2 Calculating $\frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial \hat{\boldsymbol{\mu}}_i}$

Examining figure 4.1, we see that $\mathcal{L}$ is dependent on $\hat{\boldsymbol{\mu}}_i$ through $\mathbf{d}_i$ and α_i. Therefore, we can use the chain rule to form

$$\frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial \hat{\boldsymbol{\mu}}_i} = \frac{\partial \mathbf{d}_i}{\partial \hat{\boldsymbol{\mu}}_i} \frac{\partial \alpha_i}{\partial \mathbf{d}_i} \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial \alpha_i}. \tag{4.9}$$

We expand the Gaussian power term on line 6 of algorithm 1 with

$$\begin{aligned} \mathbf{d}_i^T \hat{\Sigma}_i^{-1} \mathbf{d}_i &= [d_1, d_2] \begin{bmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \end{bmatrix} \\ &= d_1 s_{11} d_1 + d_2 s_{21} d_1 + d_1 s_{12} d_2 + d_2 s_{22} d_2 \\ &= d_1 s_{11} d_1 + 2 d_2 s_{12} d_1 + d_2 s_{22} d_2 \end{aligned} \tag{4.10}$$

where, in equation 4.10, we exploit the fact that $s_{12} = s_{21}$ in the inverse of a symmetric covariance matrix (recall that the inverse of a symmetric matrix is also symmetric). Next, we restate α_i with

$$\alpha_i = \sigma e^{-\frac{1}{2}(d_1 s_{11} d_1 + d_2 s_{21} d_1 + d_1 s_{12} d_2 + d_2 s_{22} d_2)}$$

and supply the gradients

$$\frac{\partial \mathbf{d}_i}{\partial \hat{\boldsymbol{\mu}}_i} = -\mathbf{I} \qquad \frac{\partial \alpha_i}{\partial \mathbf{d}_i} = \begin{bmatrix} -\frac{\sigma(2 s_{11} d_1 + 2 d_2 s_{12})}{2} e^{-\frac{1}{2}\mathbf{d}_i^T \hat{\Sigma}_i^{-1} \mathbf{d}_i} \\ -\frac{\sigma(2 s_{12} d_1 + 2 d_2 s_{22})}{2} e^{-\frac{1}{2}\mathbf{d}_i^T \hat{\Sigma}_i^{-1} \mathbf{d}_i} \end{bmatrix} = \begin{bmatrix} -\alpha_i(s_{11} d_1 + d_2 s_{12}) \\ -\alpha_i(s_{21} d_1 + d_2 s_{22}) \end{bmatrix}.$$

Lastly, we plug the gradients into equation 4.9 and complete our calculation:

$$\begin{aligned}\frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial \hat{\boldsymbol{\mu}}_i} &= \frac{\partial \mathbf{d}_i}{\partial \hat{\boldsymbol{\mu}}_i}\frac{\partial \alpha_i}{\partial \mathbf{d}_i}\frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial \alpha_i} \\ &= -I \cdot \begin{bmatrix} -\alpha_i(s_{11}d_1 + d_2 s_{12}) \\ -\alpha_i(s_{12}d_1 + d_2 s_{22}) \end{bmatrix} \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial \alpha_i} \\ &= \begin{bmatrix} s_{11}d_1 + d_2 s_{12} \\ s_{12}d_1 + d_2 s_{22} \end{bmatrix} \alpha_i \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial \alpha_i} \\ &= \hat{\Sigma}_i^{-1}\mathbf{d}_i \alpha_i \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial \alpha_i} \end{aligned} \tag{4.11}$$

4.4.3 Calculating $\frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial \hat{\Sigma}_i^{-1}}$

We see from figure 4.1 that $\hat{\Sigma}_i^{-1}$ only has one dependency, α_i, so

$$\frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial \hat{\Sigma}_i^{-1}} = \frac{\partial \alpha_i}{\partial \hat{\Sigma}_i^{-1}}\frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial \alpha_i}$$

and

$$\begin{aligned}\frac{\partial \alpha_i}{\partial \hat{\Sigma}_i^{-1}} &= \begin{bmatrix} \frac{\partial}{\partial s_{11}}\sigma_i \exp(-\frac{1}{2}(d_1 s_{11} d_1 + 2d_2 s_{12} d_1 + d_2 s_{22} d_2)) & \frac{\partial}{\partial s_{12}}\sigma_i \exp(-\frac{1}{2}(d_1 s_{11} d_1 + 2d_2 s_{12} d_1 + d_2 s_{22} d_2)) \\ \frac{\partial}{\partial s_{21}}\sigma_i \exp(-\frac{1}{2}(d_1 s_{11} d_1 + 2d_2 s_{12} d_1 + d_2 s_{22} d_2)) & \frac{\partial}{\partial s_{22}}\sigma_i \exp(-\frac{1}{2}(d_1 s_{11} d_1 + 2d_2 s_{12} d_1 + d_2 s_{22} d_2)) \end{bmatrix} \\ &= \begin{bmatrix} \frac{-d_1 d_1}{2}\alpha_i & -d_1 d_2 \alpha_i \\ -d_1 d_2 \alpha_i & \frac{-d_2 d_2}{2}\alpha_i \end{bmatrix} \\ &= \alpha_i \begin{bmatrix} \frac{-d_1 d_1}{2} & -d_1 d_2 \\ -d_1 d_2 & \frac{-d_2 d_2}{2} \end{bmatrix}. \end{aligned}$$

We can conclude with

$$\frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial \hat{\Sigma}_i^{-1}} = \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial \alpha_i}\alpha_i \begin{bmatrix} \frac{-d_1 d_1}{2} & -d_1 d_2 \\ -d_1 d_2 & \frac{-d_2 d_2}{2} \end{bmatrix} \tag{4.12}$$

4.4.4 Calculating $\frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial \sigma_i}$

We start with the template

$$\frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial \sigma_i} = \frac{\partial \alpha_i}{\partial \sigma}\frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial \alpha_i} \tag{4.13}$$

and, from algorithm 1, line 6, we have

$$\frac{\partial \alpha_i}{\partial \sigma} = \exp\left(-\frac{1}{2}\mathbf{d}_i^T \hat{\Sigma}_i^{-1}\mathbf{d}_i\right) = \frac{\alpha_i}{\sigma_i}.$$

Putting it all together, we have

$$\frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial \sigma_i} = \frac{\alpha_i \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial \alpha_i}}{\sigma_i}. \tag{4.14}$$

4.4.5 Calculating $\frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial \alpha_i}$

Figure 4.1 shows us that T_{i+1} and C depend on α_i. The chain rule gives us

$$\begin{aligned}\frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial \alpha_i} &= \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial C}\frac{\partial C}{\partial \alpha_i} + \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial T_{i+1}}\frac{\partial T_{i+1}}{\partial \alpha_i} \\ &= \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial C}\mathbf{c}_i(\mathbf{v})T_i - \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial T_{i+1}}T_i \end{aligned} \tag{4.15}$$

We could stop here and just track $\frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial T_i}$ through the loop like we do for T, but the implementation given by Kerbl et al. suggests that they took a slightly different approach. They probably devised a closed-form expression for $\frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial T_{i+1}}$ by unrolling it:

$$
\begin{aligned}
\frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial T_{i+1}} &= \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial T_{i+2}} \frac{\partial T_{i+2}}{\partial T_{i+1}} + \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial C} \frac{\partial C}{\partial T_{i+1}} \\
&= \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial T_{i+2}} (1 - \alpha_{i+1}) + \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial C} c_{i+1} \alpha_{i+1} \\
&= \left(\frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial T_{i+3}} (1 - \alpha_{i+2}) + \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial C} c_{i+2} \alpha_{i+2} \right) (1 - \alpha_{i+1}) + \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial C} c_{i+1} \alpha_{i+1} \\
&= \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial T_{i+3}} (1 - \alpha_{i+2})(1 - \alpha_{i+1}) + \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial C} c_{i+2} \alpha_{i+2} (1 - \alpha_{i+1}) + \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial C} c_{i+1} \alpha_{i+1} \\
&= \left(\frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial C} \frac{\partial C}{\partial T_{final}} \prod_{k=i+1}^{n} (1 - \alpha_k) \right) + \left(\frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial C} \sum_{j=i+1}^{n} \mathbf{c}_j(\mathbf{v}) \alpha_j \prod_{k=i+1}^{j-1} (1 - \alpha_k) \right) \\
&= \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial C} \left(C_{bg} \prod_{k=i+1}^{n} (1 - \alpha_k) + \sum_{j=i+1}^{n} \mathbf{c}_j(\mathbf{v}) \alpha_j \prod_{k=i+1}^{j-1} (1 - \alpha_k) \right).
\end{aligned}
$$

Plugging the unrolled $\frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial T_{i+1}}$ into line 4.15, we have

$$
\begin{aligned}
\frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial \alpha_i} &= \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial C} \mathbf{c}_i(\mathbf{v}) T_i - \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial T_{i+1}} T_i \\
&= \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial C} \mathbf{c}_i(\mathbf{v}) T_i - \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial C} \left(C_{bg} \prod_{k=i+1}^{n} (1 - \alpha_k) + \sum_{j=i+1}^{n} \mathbf{c}_j(\mathbf{v}) \alpha_j \prod_{k=i+1}^{j-1} (1 - \alpha_k) \right) T_i \\
&= T_i \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial C} \left(\mathbf{c}_i(\mathbf{v}) - \left(C_{bg} \prod_{k=i+1}^{n} (1 - \alpha_k) + \sum_{j=i+1}^{n} \mathbf{c}_j(\mathbf{v}) \alpha_j \prod_{k=i+1}^{j-1} (1 - \alpha_k) \right) \right) \\
&= T_i \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial C} \left(\mathbf{c}_i(\mathbf{v}) - \sum_{j=i+1}^{n} \mathbf{c}_j(\mathbf{v}) \alpha_j \prod_{k=i+1}^{j-1} (1 - \alpha_k) \right) - \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial C} C_{bg} T_i \prod_{k=i+1}^{n} (1 - \alpha_k) \\
&= T_i \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial C} \left(\mathbf{c}_i(\mathbf{v}) - \sum_{j=i+1}^{n} \mathbf{c}_j(\mathbf{v}) \alpha_j \prod_{k=i+1}^{j-1} (1 - \alpha_k) \right) - \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial C} C_{bg} \prod_{k=1}^{i-1} (1 - \alpha_k) \prod_{k=i+1}^{n} (1 - \alpha_k) \\
&= T_i \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial C} \left(\mathbf{c}_i(\mathbf{v}) - \underbrace{\sum_{j=i+1}^{n} \mathbf{c}_j(\mathbf{v}) \alpha_j \prod_{k=i+1}^{j-1} (1 - \alpha_k)}_{r_i} \right) - \frac{\partial \mathcal{L}_{\mathbf{p}}}{\partial C} C_{bg} \frac{T_{\text{final}}}{(1 - \alpha_i)}.
\end{aligned} \tag{4.16}
$$

In their code, Kerbl et al. track

$$
r_i = \sum_{j=i+1}^{n} \mathbf{c}_j(\mathbf{v}) \alpha_j \prod_{k=i+1}^{j-1} (1 - \alpha_k).
$$

To track r_i across loop iterations, they use a recurrence relation which can be derived with

$$\begin{aligned}
r_{i-1} &= \sum_{j=i}^{n} \mathbf{c}_j(\mathbf{v})\alpha_j \prod_{k=i}^{j-1}(1-\alpha_k) \\
&= \mathbf{c}_i(\mathbf{v})\alpha_i \prod_{k=i}^{i-1}(1-\alpha_k) + \sum_{j=i+1}^{n} \mathbf{c}_j(\mathbf{v})\alpha_j \prod_{k=i}^{j-1}(1-\alpha_k) \\
&= \mathbf{c}_i(\mathbf{v})\alpha_i + \sum_{j=i+1}^{n} \mathbf{c}_j(\mathbf{v})\alpha_j \prod_{k=i}^{j-1}(1-\alpha_k) \\
&= \mathbf{c}_i(\mathbf{v})\alpha_i + (1-\alpha_i) \sum_{j=i+1}^{n} \mathbf{c}_j(\mathbf{v})\alpha_j \prod_{k=i+1}^{j-1}(1-\alpha_k) \\
&= \mathbf{c}_i(\mathbf{v})\alpha_i + (1-\alpha_i) r_i. \qquad (4.17)
\end{aligned}$$

Using equation 4.17, we can track r_i across iterations of algorithm 5!

Bibliography

[1] Tomas Akenine-Möller et al. *Real-Time Rendering 4th Edition*. Boca Raton, FL, USA: A K Peters/CRC Press, 2018, p. 1200. ISBN: 978-1-13862-700-0.

[2] Atilim Gunes Baydin et al. *Automatic differentiation in machine learning: a survey*. 2018. arXiv: 1502.05767 [cs.SC]. URL: https://arxiv.org/abs/1502.05767.

[3] Zhiqin Chen and Hao Zhang. *Learning Implicit Fields for Generative Shape Modeling*. 2019. arXiv: 1812.02822 [cs.GR]. URL: https://arxiv.org/abs/1812.02822.

[4] DIKU Futhark Team. *futhark-server-python: Python implementation of the Futhark server protocol*. https://github.com/diku-dk/futhark-server-python. GitHub repository, accessed 2026-05-02. 2026.

[5] DIKU, Department of Computer Science, University of Copenhagen. *sorts: Sorting Implementations in Futhark*. https://github.com/diku-dk/sorts. GitHub repository. Accessed: 2026-05-07. 2026.

[6] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[7] Peter Hedman et al. "Deep Blending for Free-viewpoint Image-based Rendering". In: 37.6 (2018), 257:1–257:15.

[8] Troels Henriksen. *Personal communication regarding manual differentiation*. In our conversation about the difficulties of manual differentiation, Henriksen related a peculiar phenomenon where a change to a primal function could result in non-localized changes to the corresponding gradient function. May 2026.

[9] Troels Henriksen et al. *The Futhark Programming Language*. https://github.com/diku-dk/futhark. 2013.

[10] M. E. Jerrell. "Automatic Differentiation and Interval Arithmetic for Estimation of Disequilibrium Models". In: *Computational Economics* 10 (1997), pp. 295–316. DOI: 10.1023/A:1008633613243.

[11] Jun Jiang et al. "What is the space of spectral sensitivity functions for digital color cameras?" In: Jan. 2013, pp. 168–179. ISBN: 978-1-4673-5053-2. DOI: 10.1109/WACV.2013.6475015.

[12] Hiroharu Kato, Yoshitaka Ushiku, and Tatsuya Harada. *Neural 3D Mesh Renderer*. 2017. arXiv: 1711.07566 [cs.CV]. URL: https://arxiv.org/abs/1711.07566.

[13] Hiroharu Kato et al. *Differentiable Rendering: A Survey*. 2020. arXiv: 2006.12057 [cs.CV]. URL: https://arxiv.org/abs/2006.12057.

[14] Bernhard Kerbl et al. "3D Gaussian Splatting for Real-Time Radiance Field Rendering". In: *ACM Transactions on Graphics* 42.4 (July 2023). URL: https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/.

[15] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG]. URL: https://arxiv.org/abs/1412.6980.

[16] Arno Knapitsch et al. "Tanks and Temples: Benchmarking Large-Scale Scene Reconstruction". In: *ACM Transactions on Graphics* 36.4 (2017).

[17] Shichen Liu et al. *Soft Rasterizer: A Differentiable Renderer for Image-based 3D Reasoning*. 2019. arXiv: 1904.01786 [cs.CV]. URL: https://arxiv.org/abs/1904.01786.

[18] Nelson L. Max. "Optical Models for Direct Volume Rendering". In: *IEEE Trans. Vis. Comput. Graph.* 1 (1995), pp. 99–108. URL: https://api.semanticscholar.org/CorpusID:1005187.

[19] Ben Mildenhall et al. "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis". In: *CoRR* abs/2003.08934 (2020). arXiv: 2003.08934. URL: https://arxiv.org/abs/2003.08934.

[20] Thomas Müller et al. "Instant Neural Graphics Primitives with a Multiresolution Hash Encoding". In: *CoRR* abs/2201.05989 (2022). arXiv: 2201.05989. URL: https://arxiv.org/abs/2201.05989.

[21] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: 1912.01703 [cs.LG]. URL: https://arxiv.org/abs/1912.01703.

[22] M. Pharr, W. Jakob, and G. Humphreys. *Physically Based Rendering: From Theory to Implementation*. MIT Press, 2023. ISBN: 9780262374040. URL: https://pbr-book.org/4ed/contents.

[23] Han Qi, Tao Cai, and Xiyue Han. *Projecting Gaussian Ellipsoids While Avoiding Affine Projection Approximation*. 2024. arXiv: 2411.07579 [cs.CV]. URL: https://arxiv.org/abs/2411.07579.

[24] Scratchapixel. *An Introduction to Volume Rendering*. https://www.scratchapixel.com/lessons/3d-basic-rendering/volume-rendering-for-developers/intro-volume-rendering.html. Accessed: 2026-05-20. 2024.

[25] Kyle Simek. *Dissecting the Camera Matrix, Part 3: The Intrinsic Matrix*. Accessed: 2026-07-09. Aug. 2013. URL: https://ksimek.github.io/2013/08/13/intrinsic/.

[26] Noah Snavely, Steven Seitz, and Richard Szeliski. "Photo tourism: exploring photo collections in 3D. ACM Trans Graph 25(3):835-846". In: *ACM Trans. Graph.* 25 (July 2006), pp. 835–846. DOI: 10.1145/1141911.1141964.

[27] Andrea Tagliasacchi and Ben Mildenhall. *Volume Rendering Digest (for NeRF)*. 2022. arXiv: 2209.02417 [cs.CV]. URL: https://arxiv.org/abs/2209.02417.

[28] Zhou Wang et al. "Image quality assessment: from error visibility to structural similarity". In: *IEEE Transactions on Image Processing* 13.4 (2004), pp. 600–612. DOI: 10.1109/TIP.2003.819861.

[29] Alex Yu et al. *Plenoxels: Radiance Fields without Neural Networks*. 2021. arXiv: 2112.05131 [cs.CV]. URL: https://arxiv.org/abs/2112.05131.

[30] Richard Zhang et al. "The Unreasonable Effectiveness of Deep Features as a Perceptual Metric". In: *CoRR* abs/1801.03924 (2018). arXiv: 1801.03924. URL: http://arxiv.org/abs/1801.03924.

[31] Matthias Zwicker et al. "EWA Splatting". In: *IEEE Transactions on Visualization and Computer Graphics* (2002). URL: https://www.cs.umd.edu/~zwicker/publications/EWASplatting-TVCG02.pdf.