Photorealistic 3D Scene Rendering using Gaussian Splatting

End-to-End Implementation using Nerfstudio and COLMAP on Cloud Infrastructure

Author:

Vasco Sá

GitHub | LinkedIn

28 January 2026

Abstract

This project explores the pipeline of converting a standard 2D video into a navigable 3D scene using Gaussian Splatting. By leveraging Nerfstudio for training and React Three Fiber for rendering, I successfully created a web-based viewer that preserves the photorealistic quality of Neural Radiance Fields while enabling real-time interaction. The report analyzes the training process on cloud infrastructure (Kaggle), achieving a training time of approximately 17 minutes on NVIDIA T4 GPUs, and the optimization for web delivery, reducing the final model from 80MB to 11.2MB through aggressive alpha culling. The project examines specific challenges related to textureless surfaces in the input data, demonstrating that while objects with distinct geometry are reconstructed with high fidelity, uniform surfaces introduce artifacts requiring careful hyperparameter tuning. This work serves as a proof-of-concept for democratizing 3D capture using entirely free, open-source tools.

Keywords

Gaussian Splatting, Photogrammetry, Structure-from-Motion, Nerfstudio, Neural Radiance Fields, Digital Twin, React Three Fiber, Point Cloud Optimization, Web-based Rendering.

Acknowledgements

I would like to acknowledge the open-source contributions of the Nerfstudio Team (University of California, Berkeley) for developing the modular framework used in this project. Additionally, credit is due to Kerbl et al. (Inria, University of Côte d'Azur) for the foundational research on 3D Gaussian Splatting, which made real-time rasterization of radiance fields possible.

Special thanks to the COLMAP developers for the Structure-from-Motion algorithms that enabled the initial camera pose estimation required for this pipeline.

Table of Contents

Nomenclature

Acronym	Definition
3DGS	3D Gaussian Splatting
COLMAP	Structure-from-Motion and Multi-View Stereo (Software)
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture (NVIDIA)
FPS	Frames Per Second
GPU	Graphics Processing Unit
LiDAR	Light Detection and Ranging
MLP	Multi-Layer Perceptron
NeRF	Neural Radiance Field
R3F	React Three Fiber
RGB	Red Green Blue
SfM	Structure-from-Motion
SGD	Stochastic Gradient Descent
SH	Spherical Harmonics
SIFT	Scale-Invariant Feature Transform
VRAM	Video Random Access Memory

1. Introduction

1.1 Context and Motivation

Traditionally, creating high-fidelity 3D environments required either labor-intensive manual modeling or expensive laser scanning (LiDAR) equipment.

In recent years, the field of Computer Vision has shifted towards Neural Rendering. The introduction of Neural Radiance Fields (NeRFs) in 2020 marked a significant breakthrough, allowing AI to "learn" a 3D scene from 2D images. However, NeRFs suffer from high computational costs due to volumetric ray-marching, making them unsuitable for real-time rendering on consumer devices.

3D Gaussian Splatting (3DGS), introduced by Kerbl et al. in 2023, addresses this limitation. By representing a scene as a collection of anisotropic 3D Gaussians rather than a continuous field or a polygon mesh, 3DGS allows for photorealistic quality with real-time rendering speeds via rasterization. This project explores the practical application of this technology to democratize 3D capture.

1.2 Problem Statement

While 3DGS offers superior performance, the pipeline remains fragmented and technically demanding. Training these models typically requires high-end local GPUs (e.g., NVIDIA RTX 3090/4090) with massive VRAM, and the resulting files are often too large or incompatible with standard web standards.

Furthermore, relying on consumer-grade input data (such as handheld smartphone video) introduces noise, motion blur, and textureless surfaces that traditional Structure-from-Motion (SfM) algorithms struggle to process. There is a need for a streamlined, accessible workflow that bridges the gap between raw smartphone footage and a lightweight, browser-ready 3D experience.

1.3 Objectives

The primary objective of this project is to design and implement an end-to-end pipeline for Neural Rendering, demonstrating that high-quality 3D scenes can be created and deployed using free cloud resources and standard web technologies.

Specific goals include:

1. **Automated Processing:** Implement a "headless" Structure-from-Motion pipeline using COLMAP to extract camera poses from standard video files.
2. **Cloud-Based Training:** Leverage Kaggle’s NVIDIA T4 GPUs to train a Gaussian Splatting model (Nerfstudio’s splatfacto) without local hardware dependencies.
3. **Web Optimization:** Develop a conversion strategy to transform raw Nerfstudio checkpoints (.ply) into optimized binary formats (.splat) suitable for web streaming.
4. **Interactive Visualization:** Build a high-performance web viewer using React Three Fiber to render the scene in real-time within a modern browser.

1.4 Project Scope

This report documents the complete lifecycle of the "3DGS Viewer" project. It covers the data acquisition methodology, the mathematical foundations of the training process, and the engineering challenges encountered during cloud deployment.

Specifically, this project functions as a proof-of-concept validation for the 3DGS pipeline. Rather than a large-scale architectural scan, the scope is limited to a micro-scale reconstruction: capturing a distinct foreground object (a plastic pen) placed on a challenging, uniform background surface (a grey carpet). This setup serves as a stress test for the algorithm's ability to resolve fine details and handle surface ambiguity. The final output is a functional web application that allows users to interactively inspect the reconstructed 3D scene in a modern browser.

2. Theoretical Background

2.1 Structure-from-Motion (SfM)

The first step in any photogrammetry or neural rendering pipeline is determining the camera's position in 3D space for every frame of the input video. Since standard video does not contain depth information, I utilize Structure-from-Motion (SfM).

I employed COLMAP, a state-of-the-art SfM library, which performs two key operations:

1. **Feature Extraction:** It detects distinct visual features (keypoints) in each image using algorithms like SIFT (Scale-Invariant Feature Transform).
2. **Feature Matching:** It tracks these features across multiple frames. By analyzing the parallax (how much a point moves between frames), it triangulates the 3D position of the point and the 6-DoF (Degrees of Freedom) pose of the camera (as illustrated in Figure 1).

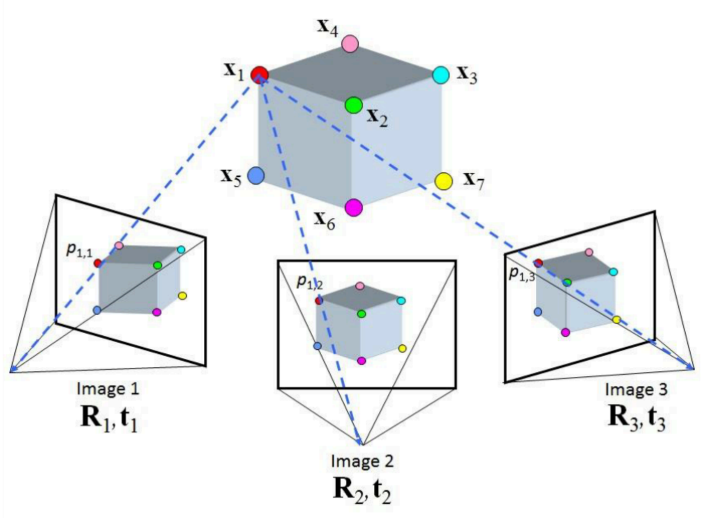


Figure 1: *The Structure-from-Motion (SfM) process. Multiple 2D images are analyzed to triangulate 3D point positions (xn) by calculating individual camera rotation (R) and translation (t) matrices. (Source: Yilmaz & Karakus, 2013).*

The output of this stage is a sparse point cloud and a set of camera matrices (intrinsics and extrinsics), which serve as the "ground truth" geometry for training the model.

2.2 Neural Radiance Fields (NeRFs)

To understand the innovation of Gaussian Splatting, it is necessary to understand its predecessor: Neural Radiance Fields (NeRFs).

NeRFs represent a scene as a continuous volumetric field using a Multi-Layer Perceptron (MLP). To render a single pixel, the algorithm shoots a ray from the camera into the scene and samples color and density at hundreds of points along that ray. This process is known as Volumetric Ray-Marching.

$$C(r) = \int_{t_n}^{t_f} T(t) \cdot \sigma(r(t)) \cdot c(r(t), d) dt$$

Where:

- C(r) is the final expected color of the pixel.
- σ(r(t)) represents the Volume Density (opacity) at point t.
- c(r(t),d) is the view-dependent RGB Color emitted at that point.
- T(t) is the Transmittance, representing the probability that the ray travels from the camera to point t without being blocked by clearer geometry (accumulated transparency).

While NeRFs produce high-quality images, this integration process is computationally expensive. Computationally, evaluating this integral requires approximating it as a discrete sum, forcing the algorithm to query the neural network to sample σ and c hundreds of times for every single pixel. This results in millions of neural network evaluations to render a single frame, making real-time rendering on consumer hardware nearly impossible without massive downscaling.

2.3 3D Gaussian Splatting (3DGS)

3D Gaussian Splatting (3DGS), introduced by Kerbl et al. (2023) in their paper ‘3D Gaussian Splatting for Real-Time Radiance Field Rendering’, addresses this limitation. By representing a scene as a collection of anisotropic 3D Gaussians rather than a continuous field or a polygon mesh, 3DGS achieves photorealistic rendering quality comparable to NeRFs with real-time rendering speeds via rasterization.

2.3.1 The Gaussian Primitive

Instead of a polygon mesh (triangles) or a voxel grid, the scene is composed of millions of "splats." Each splat is defined as a 3D Gaussian parameterized by a mean position μ (center) and a 3D covariance matrix Σ:

$$G(x) = e^{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)}$$

The covariance matrix (Σ) controls the shape and orientation of the Gaussian. Crucially, 3DGS does not optimize Σ directly, as it must remain positive semi-definite to represent a valid physical volume. Instead, the model factorizes it into two learnable components: Scaling (S) and Rotation (R).

$$\Sigma = RSS^TR^T$$

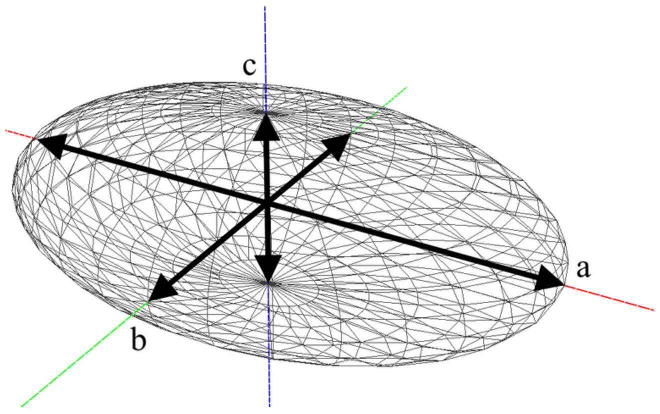


Figure 2: *The 3D Gaussian Primitive. A triaxial ellipsoid model where axes a, b, and c represent the learned scaling parameters. These axes correspond to the diagonal elements of the scaling matrix (S), defining the shape and orientation of the splat (Source: Bandyopadhyay et al., 2021).*

Where:

- **S (Scaling)**: A diagonal matrix containing three scaling factors (sx,sy,sz). These correspond to the axes a, b, and c illustrated in Figure 2, allowing the Gaussian to stretch into an ellipsoid (anisotropy).

$$S = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}$$

- **R (Rotation)**: A 3×3 rotation matrix derived from a learned quaternion. This allows the ellipsoid to rotate in 3D space to align with the surface geometry of the object.

$$R = \begin{bmatrix} 1-2(y^2+z^2) & 2(xy-rz) & 2(xz+ry) \\ 2(xy+rz) & 1-2(x^2+z^2) & 2(yz-rx) \\ 2(xz-ry) & 2(yz+rx) & 1-2(x^2+y^2) \end{bmatrix}$$

2.3.2 Rasterization

The key performance breakthrough of 3DGS is its rendering method. Instead of marching rays, it uses Rasterization:

1. **Projection:** The 3D Gaussians are projected onto the 2D image plane.
2. **Sorting:** The splats are sorted by depth (distance from the camera).
3. **Alpha Blending:** The renderer processes the sorted splats from front-to-back, accumulating color and opacity until the pixel is opaque.

This approach is computationally similar to standard video game rendering (rasterizing triangles), enabling real-time interaction on consumer hardware while preserving the photorealistic quality of NeRFs. Performance varies by GPU capability and scene complexity, with modern gaming GPUs capable of 60+ FPS while mid-range hardware maintains interactive frame rates.

2.3.3 Adaptive Density Control

One of the key advantages of 3DGS over static meshes is its ability to modify the geometry during training. The model starts with a sparse set of points (from Structure-from-Motion) and dynamically densifies or prunes them based on the gradient of the loss function.

- **Densification:** Every 100 iterations, the model identifies Gaussians with large position gradients (indicating high error or missing geometry).
    - **Clone (Under-Reconstruction):** If a Gaussian is small, it is cloned to cover the area better.
    - **Split (Over-Reconstruction):** If a Gaussian is too large, it is split into two smaller Gaussians.

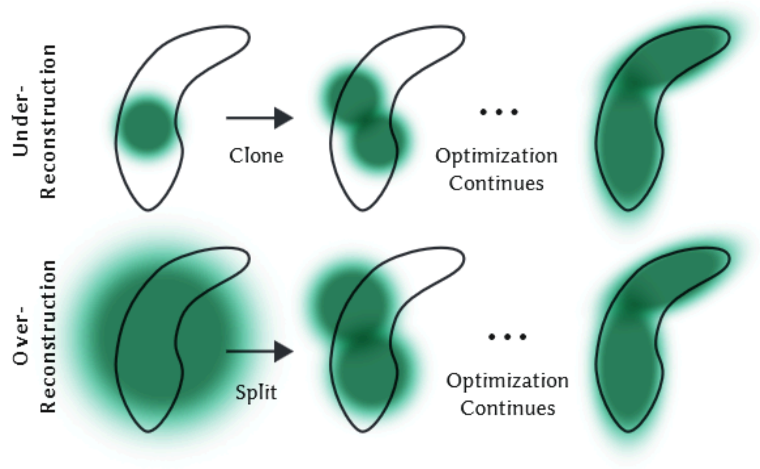


Figure 3: *Adaptive density control strategies used during optimization. The model creates new Gaussians by cloning in under-reconstructed regions (top) and splitting in over-reconstructed regions (bottom). (Source: Kerbl et al. (2023).*

- **Pruning:** Gaussians that are virtually transparent (low opacity) or extremely large are removed to prevent "floaters" and reduce computational cost.

This mechanism allows the model to allocate more resources to complex areas (like the pen) and fewer to flat regions (like the surface).

2.3.4 Spherical Harmonics (View-Dependent Color)

To achieve photorealism, 3DGS does not store a single static RGB color for each point. Instead, it utilizes Spherical Harmonics (SH).

SH functions are a series of basis functions defined on the surface of a sphere. They allow the color of a Gaussian to change depending on the viewing angle. This enables the capture of view-dependent effects such as specular highlights (shininess) and reflections.

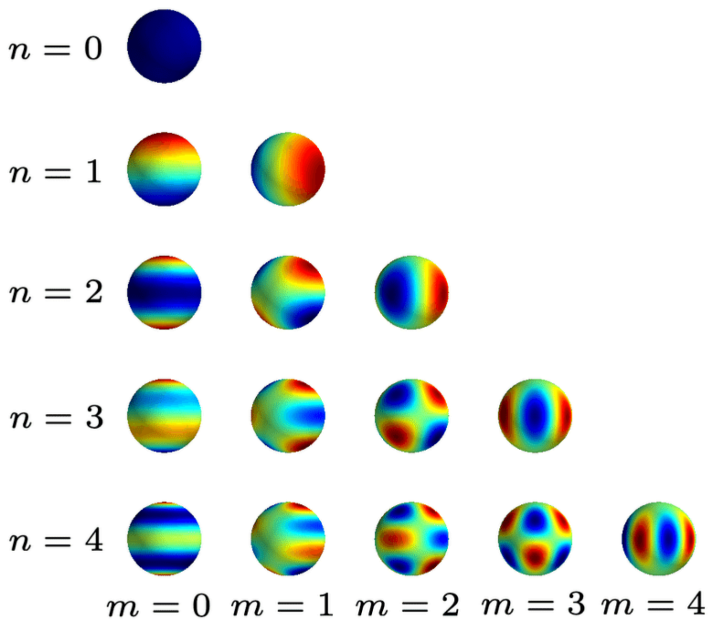


Figure 4: *Visualization of Spherical Harmonic basis functions. These mathematical "shapes" allow each Gaussian to store color data that changes based on the viewing angle, enabling the model to reconstruct realistic specular highlights and reflections.(Source: Hollebon & Fazi, 2020)*

SH Degree (n)	**Coefficients per Color Channel**	**Total Coefficients (RGB)**	**Impact on File Size**
0	1	3	**Minimal:** Base color only (no reflections).
1	4	12	**Low:** Basic directional lighting.
2	9	27	**Moderate:** Improved specular highlights.
3 (Nerfstudio Default)	16	48	**High:** Professional-grade reflections/specularity.

Transitioning from Degree 0 to Degree 3 increases the color data storage by 16x. In scenes with millions of Gaussians, this can be the difference between a 50MB file (web-friendly) and a 500MB file (slow to load).

2.3.5 The Loss Function

The model is trained by minimizing the difference between the rendered image and the original video frame. The loss function (L) is a combination of two metrics:

1. **L1 Loss:** Measures the absolute pixel difference (Manhattan distance).
2. **LD-SSIM (Structural Similarity Index):** Ensures that the perceived structure and texture of the image match the ground truth, rather than just individual pixel values.

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

The splatfacto architecture utilizes a default structural weight parameter of λ=0.2. This configuration assigns 20% of the optimization focus to structural integrity (SSIM) and 80% to raw color accuracy (L1). This baseline balance effectively prioritized accurate color reproduction while maintaining sufficient structural constraints to preserve the sharp edges of the pen against the textured background.

2.3.6 Optimization via Gradient Descent

To minimize the defined loss function (L), the model employs Stochastic Gradient Descent (SGD), specifically utilizing the Adam optimizer. During each training iteration, the system calculates the "gradient" - a mathematical vector representing the direction and magnitude of error for every parameter of the millions of Gaussians (position, opacity, scale, and rotation).

Through backpropagation, the optimizer iteratively nudges these parameters in the opposite direction of the gradient to reduce the total error. This iterative process allows the model to "learn" the scene’s geometry and appearance, gradually refining the sparse point cloud into a photorealistic reconstruction over thousands of steps.

3. Methodology

3.1 Data Acquisition (Micro-Scale Reconstruction)

The experiment focused on a micro-scale reconstruction of a specific foreground object: a pen placed on a highly textured carpet. Data was captured using a handheld smartphone camera, recording an 11-second orbital video moving in a 360-degree arc around the subject to ensure omnidirectional coverage. This specific setup was chosen as a stress test for the pipeline; while the pen provided distinct geometric features (edges and specularity), the background carpet featured a uniform, repetitive texture pattern.

3.2 The Training Pipeline

The computational workload was offloaded to the Kaggle cloud platform, utilizing NVIDIA T4 GPUs to handle the high VRAM demands of Gaussian Splatting. The pipeline was built on a PyTorch backbone, which serves as the primary deep learning framework for performing the gradient descent required to optimize the Gaussian parameters. To ensure the hardware could execute these complex mathematical operations in a reasonable timeframe, I utilized CUDA 11.8, allowing the training process to run on the GPU's parallel cores rather than the CPU. The pipeline was executed in three distinct stages:

3.2.1 Preprocessing (Structure-from-Motion)

The first step involved converting the raw video into a format suitable for AI training. I used FFmpeg to deconstruct the 11-second video into individual high-resolution image frames. These frames were then processed by COLMAP via the Nerfstudio `ns-process-data` wrapper to determine camera poses. Since Kaggle is a headless cloud environment, the command was wrapped in `xvfb` (X Virtual Framebuffer) to simulate a display and prevent the software's GUI components from crashing the session.

3.2.2 Hyperparameter Configuration and Model Training

I utilized the splatfacto model with specific overrides designed to optimize the output for web streaming and ensure stability on the Kaggle cloud environment. During this phase, PyTorch iteratively updated the position, rotation, scale, and opacity of millions of Gaussians.

Hyperparameter	**Value**	**Justification**
`cull_alpha_thresh`	**0.01**	Increased from default (0.005) to aggressively remove semi-transparent "haze." This was the primary driver for reducing file size to 11.2MB.
`stop_split_at`	**10,000**	Halted geometry creation at step 10,000 (of 15,000) to dedicate the final 33% of training purely to color refinement.
`max_num_iterations`	**15,000**	Reduced training duration to ~17 minutes, sufficient for convergence on micro-scale scenes while minimizing GPU hours.
`MAX_JOBS`	**5**	Enforced environment variable constraint to cap dataloading workers, preventing RAM usage from exceeding Kaggle's 30GiB session limit.

3.2.3 Post-Processing for Real-Time Deployment

The standard output from Nerfstudio is a proprietary checkpoint format. I first exported this to a standard `.ply` (Polygon File Format) point cloud using `ns-export`. However, raw `.ply` files are inefficient for real-time web streaming, often exceeding 100MB in size. To resolve this, I processed the file with a custom Python script (`convert.py`) to serialize the data into a compressed binary `.splat` format. This conversion reorganizes the Gaussian attributes - position, scale, rotation, and color - into a linear buffer optimized for rapid GPU sorting in the browser.

3.3 Web Implementation

To visualize the result, I developed a frontend application using React 18 and React Three Fiber (R3F). The application uses the `@react-three/drei` library to asynchronously load the binary `.splat` file. The viewer is configured with an `OrbitControls` camera system, allowing users to rotate around the center of the reconstructed object and zoom in to inspect high-frequency details such as the specular highlights on the pen casing.

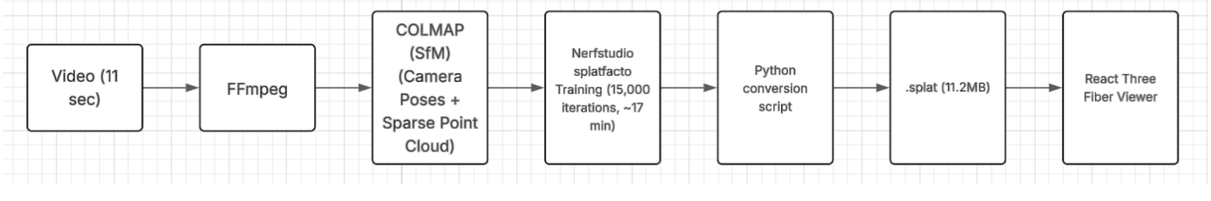

Video (11 sec)
FFmpeg
COLMAP (SfM) (Camera Poses + Sparse Point Cloud)
Nerfstudio splatfacto Training (15,000 iterations, ~17 min)
Python conversion script
.splat (11.2MB)
React Three Fiber Viewer

Figure 5: *End-to-end Gaussian Splatting pipeline. The process converts an 11s video into a 11.2MB interactive 3D scene, with training completed in ~17 minutes on cloud GPUs.*

4. Results and Analysis

4.1 Visual Fidelity

I successfully reconstructed the target object (the pen) with high geometric accuracy. The sharp edges of the pen casing were preserved, and the Spherical Harmonics effectively captured the specular highlights on the plastic surface. This demonstrates the model's ability to handle view-dependent lighting effects.

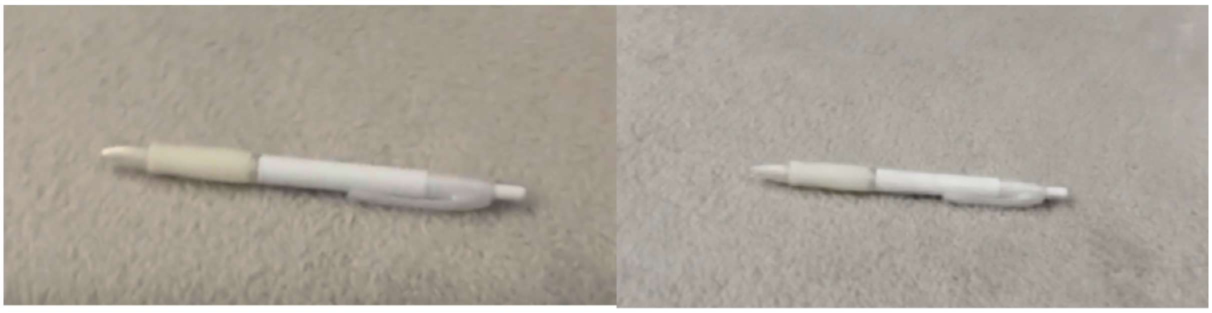


Figure 6: *Comparison between a single extracted video frame (left) and the rendered Gaussian Splat (right). The reconstruction accurately captures the material response of the plastic object (the pen), particularly the shifting specular reflections driven by the view-dependent Spherical Harmonics coefficients*

However, I observed that the background reconstruction quality was variable; while the general planar structure of the carpet was preserved, the surface texture exhibited significant noise where the feature extraction was sparse.

4.2 Performance Metrics

The final optimized model demonstrated high efficiency for web deployment. I reduced the original `.ply` point cloud from approximately 80 MB to a final binary `.splat` asset of 11.2 MB by applying alpha culling.

4.3 Artifact Analysis

A notable issue identified in the final render is the presence of high-frequency "floaters" - semi-transparent, phantom geometries that appear around the subject and near the surface plane. These artifacts are a direct consequence of the model attempting to explain visual inconsistencies in the training data where the geometric ground truth was ambiguous (see Figure 6).

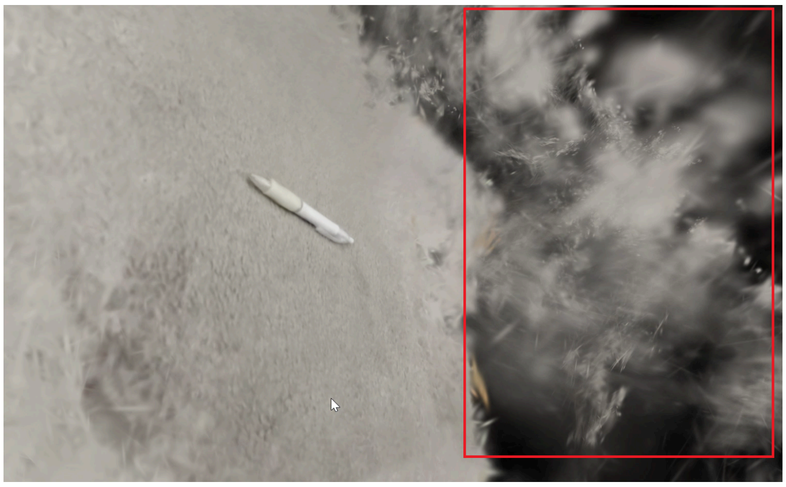


Figure 7: *Reconstruction artifacts ("floaters") identified in the final render. The red highlighted area shows phantom geometry generated by the model*

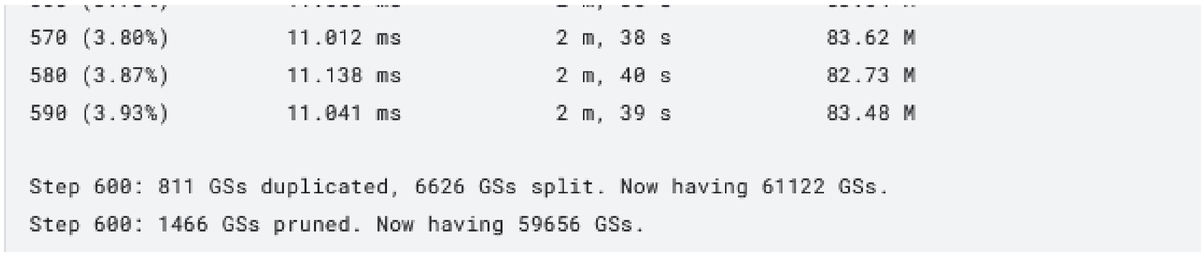
This noise was primarily driven by the low-texture surface of the uniform grey carpet, which lacked the distinct feature points necessary for COLMAP to perform accurate triangulation. This was further compounded by motion blur from the handheld camera jitter, which reduced the sharpness required for precise pixel matching across frames. Finally, lighting inconsistencies, such as shifting shadows caused by the camera's movement, violated the static scene assumption of the 3DGS algorithm. Because the pixel intensities changed depending on the viewing angle, the model generated these "floating" splats as a mathematical workaround to account for the fluctuating light and blurred details.

4.4 Analysis of Training Logs and Optimization

Because the training was conducted in a headless cloud environment on Kaggle, I monitored the evolution of the model through real-time telemetry and training logs rather than a live 3D viewer. These logs provided the necessary evidence that the optimization process was behaving as intended.

4.4.1 Adaptive Densification and Pruning

The structural evolution of the scene was driven by an adaptive densification scheme. The log confirms that the model underwent its first major structural refinement at Step 600, where 811 Gaussians were duplicated (cloned) and 6,626 were split, resulting in a jump to 59,656 Gaussians (as shown in Figure 8).

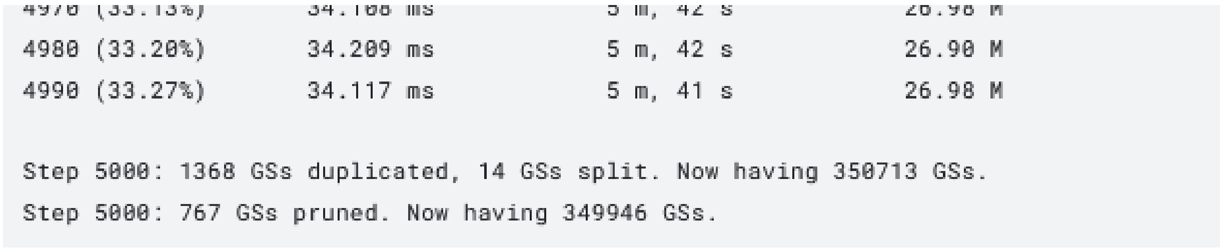


```
570 (3.80%)        11.012 ms          2 m, 38 s          83.62 M
580 (3.87%)        11.138 ms          2 m, 40 s          82.73 M
590 (3.93%)        11.041 ms          2 m, 39 s          83.48 M

Step 600: 811 GSs duplicated, 6626 GSs split. Now having 61122 GSs.
Step 600: 1466 GSs pruned. Now having 59656 GSs.
```

Figure 8: *Console output at Step 600 demonstrating the initiation of adaptive density control, where the model significantly expanded its primitive count to capture finer geometric details.*

By Iteration 5,000, the primitive count had expanded to 349,946 Gaussians, representing a 552% increase from the initial state. At this stage, the optimizer also pruned 767 transparent splats, maintaining a lean memory footprint.



```
4970 (33.13%)      34.166 ms          5 m, 42 s          26.98 M
4980 (33.20%)      34.209 ms          5 m, 42 s          26.90 M
4990 (33.27%)      34.117 ms          5 m, 41 s          26.98 M

Step 5000: 1368 GSs duplicated, 14 GSs split. Now having 350713 GSs.
Step 5000: 767 GSs pruned. Now having 349946 GSs.
```

Figure 9: *Intermediate reconstruction at Iteration 5,000. This densification allows for the emergence of high-frequency details on the pen casing and the initial formation of specular highlights, as the model successfully identifies and clones Gaussians in high-gradient regions.*

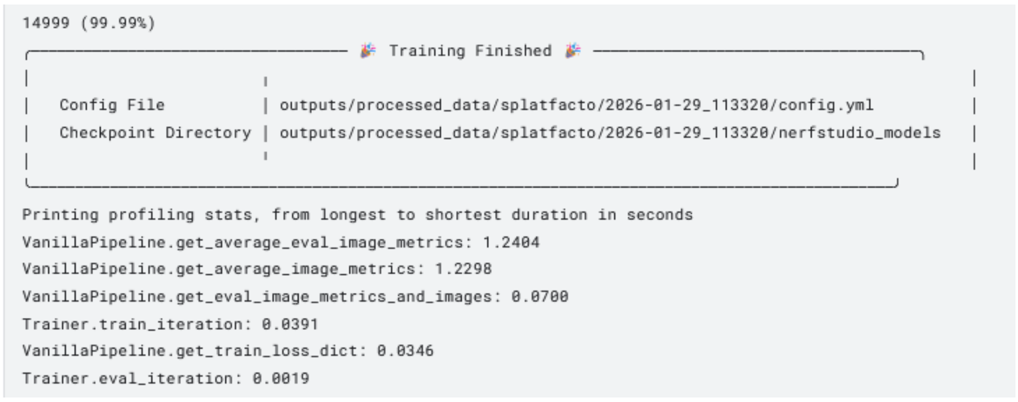
As per the methodology, all new splat creation was halted at 10,000 steps to focus purely on refining existing geometry.

4.4.2 Training Efficiency and Convergence

The training process demonstrated remarkable efficiency, successfully concluding at 15,000 iterations (Step 0 to 14,999). A detailed timing summary derived from the execution logs provides the following performance benchmarks:

- **Throughput and Velocity:** The pipeline exhibited a standard "warm-up" curve, where initial steps suffered from high latency (Step 0: ~9.26s) due to CUDA kernel compilation and initial VRAM allocation. However, the system rapidly accelerated, achieving a converged iteration velocity of approximately 69 ms for the vast majority of the training duration.
- **Total Duration:** By extrapolating this stable-state latency (15,000 steps × ~69 ms), the total training duration is calculated to be approximately 17 minutes. This aligns with the rapid turnaround expected of the 3DGS rasterization pipeline, which avoids the lengthy volumetric sampling of traditional NeRFs.
- **Computational Profiling:** Internal profiling statistics reveal that the core optimization step (`Trainer.train_iteration`) was even faster, averaging just 39.1 ms. The discrepancy between the core compute time (39 ms) and the total wall-clock time (69 ms) indicates that approximately 43% of the runtime was attributed to Python-based overhead, specifically data loading and validation metric calculation.
- **Data Processing Rate:** The average training throughput stabilized between 14 - 15 million rays per second, peaking at 24.36 million rays per second during high-efficiency cycles.

This rapid convergence highlights the primary advantage of the 3DGS pipeline over traditional NeRF methods; by utilizing a point-based rasterization approach instead of volumetric ray-marching, the model achieved photorealism in minutes rather than hours.

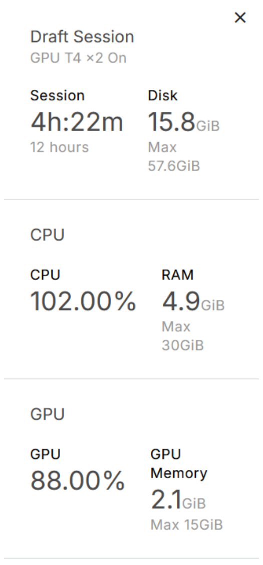


```
14999 (99.99%)
╭──────────────────────────────── 🎉 Training Finished 🎉 ────────────────────────────────╮
│                     ╷                                                                    │
│   Config File       │ outputs/processed_data/splatfacto/2026-01-29_113320/config.yml     │
│   Checkpoint Directory │ outputs/processed_data/splatfacto/2026-01-29_113320/nerfstudio_models │
│                     ╵                                                                    │
╰──────────────────────────────────────────────────────────────────────────────────────────╯
Printing profiling stats, from longest to shortest duration in seconds
VanillaPipeline.get_average_eval_image_metrics: 1.2404
VanillaPipeline.get_average_image_metrics: 1.2298
VanillaPipeline.get_eval_image_metrics_and_images: 0.0700
Trainer.train_iteration: 0.0391
VanillaPipeline.get_train_loss_dict: 0.0346
Trainer.eval_iteration: 0.0019
```

Figure 10: *Profiling statistics from the training log. The metric `Trainer.train_iteration` (0.0391s) indicates that the GPU optimization loop ran at approximately 25 steps per second. The discrepancy between this and the total iteration time (~69ms) highlights the overhead of Python-based data loading and metric evaluation.*

4.4.3 GPU Resource Stability

I utilized the Kaggle session monitor to ensure the NVIDIA T4 GPU remained stable. During the active training phase, the GPU Memory (VRAM) was maintained at 2.1 GiB, which is well below the 15 GiB maximum limit. The GPU utilization averaged 88%, demonstrating that the workload was effectively parallelized across the CUDA cores. This stable resource footprint confirms that the optimization constraints I implemented (such as `MAX_JOBS=5`) successfully prevented memory overflows.



Draft Session
GPU T4 ×2 On

Session
4h:22m
12 hours

Disk
15.8GiB
Max
57.6GiB

CPU

CPU
102.00%

RAM
4.9GiB
Max
30GiB

GPU

GPU
88.00%

GPU Memory
2.1GiB
Max 15GiB

Figure 11: *Kaggle session telemetry. While CPU and RAM usage remained stable, the GPU utilization peaked at 88% with a 2.1 GiB VRAM footprint, validating the efficiency of the splatfacto model on cloud infrastructure.*

5. Discussion

5.1 Challenges in Cloud Training

Deploying the training pipeline on a cloud environment like Kaggle presented specific engineering challenges. The primary obstacle was the "headless" nature of the server, which lacks a physical display interface. Tools like COLMAP, which often rely on GUI components, required the use of X Virtual Framebuffer (`xvfb`) to simulate a monitor and prevent runtime crashes. Additionally, managing VRAM on the NVIDIA T4 GPUs required strict process control; I restricted the number of parallel workers (`MAX_JOBS=5`) to prevent the dataloading process from exceeding the 30GiB RAM limit, which would otherwise terminate the session.

5.2 Limitations

The current pipeline is limited by the static nature of the lighting. Unlike a mesh with a distinct material and texture map, the lighting in a Gaussian Splat is "baked" into the model via Spherical Harmonics. This means the lighting conditions cannot be changed dynamically in the web viewer. Furthermore, while the `.splat` format is highly optimized for rendering, the file size (11 MB) is still significantly larger than a typical low-poly game asset, potentially impacting the user experience on slower mobile networks.

6. Conclusion and Future Work

This project functions as a successful proof-of-concept for a low-cost, end-to-end Neural Rendering pipeline. I demonstrated that it is possible to transform a simple smartphone video into an interactive, photorealistic 3D web experience using entirely open-source tools and free cloud compute resources. The "micro-scale" stress test revealed that while objects with distinct geometry (the pen) are reconstructed with high fidelity, textureless or repetitive surfaces (the carpet) introduce specific artifacts that require aggressive hyperparameter tuning to mitigate.

Future work would focus on implementing a post-processing cleaning step. Tools such as "SuperSplat" allow for the manual deletion of floating artifacts from the `.splat` file before deployment. Additionally, implementing a progressive loading system in the React viewer would improve the initial user experience, allowing a lower-quality version of the scene to appear instantly while the high-resolution data streams in the background.

References

- **Kerbl, B., Kopanas, G., Leimkühler, T., & Drettakis, G. (2023).** *3D Gaussian Splatting for Real-Time Radiance Field Rendering.* ACM Transactions on Graphics.
- **Yilmaz, O., & Karakus, F. (2013).** *Stereo and kinect fusion for continuous 3D reconstruction and visual odometry*. Conference Paper.
- **Bandyopadhyay, S., Villa, J., Osmundson, A., & Nesnas, I. (2021).** *Light-Robust Pole-from-Silhouette Algorithm and Visual-Hull Estimation for Autonomous Optical Navigation to an Unknown Small Body*. Conference Paper.
- **Hollebon, J., & Fazi, F. M. (2020).** *Efficient HRTF Representation Using Compact Mode HRTFs*. Preprint, ResearchGate.
- **Tancik, M., et al. (2023).** *Nerfstudio: A Modular Framework for Neural Radiance Field Development.* arXiv preprint.
- **Schonberger, J. L., & Frahm, J. M. (2016).** *Structure-from-Motion Revisited.* IEEE Conference on Computer Vision and Pattern Recognition (CVPR).
- **Mildenhall, B., et al. (2020).** *NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis.* ECCV.