Comparative Analysis of NeRF and Gaussian Splatting for 3D Object Rendering

Suyash Ajit Chavan (IMT2021048)
3D Vision

***Abstract*—This project explores 3D object reconstruction using mobile-captured 2D images. A point cloud is generated through Structure-from-Motion (SfM), followed by 3D rendering using Neural Radiance Fields (NeRF) and Gaussian Splatting. A comparative analysis evaluates rendering quality, efficiency, and computational demands. The findings highlight the strengths and applicability of these methods for realistic 3D rendering.**

GitHub Repository: 3D_Rendering

I. Introduction

The ability to reconstruct and render 3D representations of real-world objects has profound implications in fields such as virtual reality, gaming, digital heritage preservation, and robotics. Traditional **3D modeling** approaches often rely on specialized hardware, such as LiDAR or depth sensors, which can be expensive and inaccessible for many users. Recent advances in computer vision and machine learning offer the potential to create high-quality 3D reconstructions using only 2D images captured from standard devices like mobile cameras. This project leverages **Structure-from-Motion (SfM)** for point cloud generation and compares two cutting-edge rendering techniques: **Neural Radiance Fields (NeRF)** and **Gaussian Splatting**.

Structure-from-Motion (SfM) forms the foundational step in this workflow, enabling the creation of a 3D point cloud from a set of overlapping 2D images. This point cloud serves as the intermediate representation, encapsulating the spatial structure of the object. The subsequent stages of the pipeline utilize neural rendering techniques. **NeRF** is a neural representation framework that learns to generate high-quality volumetric renderings, capturing intricate lighting effects and fine details. **Gaussian Splatting**, on the other hand, is a newer method that offers a unique approach to 3D representation by rendering point-based approximations with Gaussian kernels, often with enhanced speed and scalability.

This report presents a comprehensive analysis of the two rendering techniques, emphasizing rendering quality, computational efficiency, and usability in different scenarios. By comparing **NeRF** and **Gaussian Splatting**, this study provides insights into their respective strengths, limitations, and potential applications. Additionally, visualizations of the rendered 3D models highlight the practical outcomes of the implemented methods. The findings underscore the growing accessibility and capability of modern **3D rendering** technologies for realistic object reconstruction.

II. Dataset Creation

Creating a dataset for 3D object reconstruction involves capturing multiple 2D images from various angles. This section discusses two approaches: using multiple 2D images and using video, each with its advantages for accurate point cloud generation.

A. *Using Multiple 2D Images:*

- Multiple 2D images are captured from different perspectives of the object to provide complete coverage.
- The images should overlap to ensure feature matching and accurate reconstruction.
- This method is straightforward but time-consuming, as it requires careful planning of angles and distances.
- The quality of the generated point cloud depends on factors like the number of images, their quality, and the level of overlap.

Fig. 1. Bottle Image

B. *Using Video:*

- Video captures a large number of images automatically from a single session.
- Continuous motion of the camera ensures a large number of overlapping images, improving feature matching.
- The dynamic nature of video provides smoother and more consistent coverage of the object.
- Video helps reduce occlusions and provides better-quality datasets due to the continuous capture of the object from multiple angles.

- This method is efficient and creates a dense dataset, which improves the quality of point cloud generation and subsequent 3D rendering.

```
import cv2
import os
import math

def extract_frames(video_path, output_dir,
    standard_frames):
    # Ensure the output directory exists
    os.makedirs(output_dir, exist_ok=True)

    # Open the video file
    video_capture = cv2.VideoCapture(video_path
        )

    if not video_capture.isOpened():
        raise FileNotFoundError(f"Unable to
            open video file: {video_path}")

    # Get video properties
    total_frames = int(video_capture.get(cv2.
        CAP_PROP_FRAME_COUNT))

    if total_frames < standard_frames:
        raise ValueError(f"Video has only {
            total_frames} frames, fewer than
            the requested {standard_frames}
            frames.")

    # Calculate frame interval to get standard
        number of frames
    frame_interval = math.floor(total_frames /
        standard_frames)

    # Extract and save frames
    saved_frames = 0
    frame_index = 0
    while saved_frames < standard_frames:
        video_capture.set(cv2.
            CAP_PROP_POS_FRAMES, frame_index)
        success, frame = video_capture.read()

        if not success:
            break

        # Save the frame as an image
        frame_filename = os.path.join(
            output_dir, f"frame_{saved_frames +
             1:03d}.jpg")
        cv2.imwrite(frame_filename, frame)

        saved_frames += 1
        frame_index += frame_interval

    video_capture.release()
    print(f"Extracted {saved_frames} frames and
         saved to {output_dir}")

video_path = r"C:\Users\suyash\IIITB\3
    D_vision_final_project\bottle_video.mp4"
output_dir = r"C:\Users\suyash\IIITB\3
    D_vision_final_project\images\bottle"
standard_frames = 200  # number of frames we
    want

extract_frames(video_path, output_dir,
    standard_frames)
```

The following video is used to create multiple images (click on video to start):

III. 3D Point Cloud Generation

The generation of a **3D point cloud** from a set of 2D images is a critical step in the process of creating realistic 3D models. Point clouds serve as an intermediate representation of the spatial structure of the object, capturing its shape and depth from various perspectives. This section discusses three popular tools for 3D point cloud generation: **Colmap**, **OpenSfM**, and **Agisoft Metashape**.

Each of these tools utilizes **Structure-from-Motion (SfM)** techniques to process overlapping 2D images, extracting key features, performing image matching, and reconstructing the 3D coordinates of the object. The choice of tool may vary depending on factors such as ease of use, computational resources, and the required accuracy for subsequent rendering tasks. The following subsections provide detailed insights into the setup, point cloud generation process, and results for each tool.

A. Using Colmap:

1. Setup and Installation:

Colmap is an open-source Structure-from-Motion (SfM) and Multi-View Stereo (MVS) tool. To install:

- Download the Colmap binaries from the official website or compile it from source if required.
- Ensure dependencies like CUDA (for GPU acceleration) and CMake are installed on your system.
- Follow the installation guide to set up Colmap, and verify the installation by running `colmap -h` in the terminal.

2. 3D Point Cloud Generation:

- Capture a series of overlapping 2D images of the object from various angles.
- Use Colmap's GUI or command-line interface to perform feature extraction and matching.
- Run the SfM pipeline to generate the sparse point cloud.
- Optionally, use the MVS module to densify the point cloud.

3. Outputs/Results:

- The output includes a `points3D.txt` file containing the 3D points and optional `.ply` or `.obj` files for visualization.
- Save the results for use in rendering pipelines like NeRF and Gaussian Splatting.

B. Using OpenSfm:

1. Setup and Installation:

OpenSfM is a Python-based library for SfM workflows. To install:

- Clone the OpenSfM repository from its GitHub page.
- Set up the Python environment with dependencies using `pip install -r requirements.txt`.

- Ensure that your system has Python and necessary libraries like NumPy, OpenCV, and SciPy installed.

2. 3D Point Cloud Generation:

- Prepare a directory containing the captured 2D images of the object.
- Run the OpenSfm pipeline by configuring the `config.yaml` file and executing the pipeline script.
- The software automatically handles feature detection, matching, and reconstruction.

3. Outputs/Results:

- The output consists of a sparse point cloud in formats like `.ply` or `.json`.
- These outputs are compatible with various 3D rendering tools for further processing.

4. Experiments:

- openSfm 3D Point cloud : PointSize = 0.1

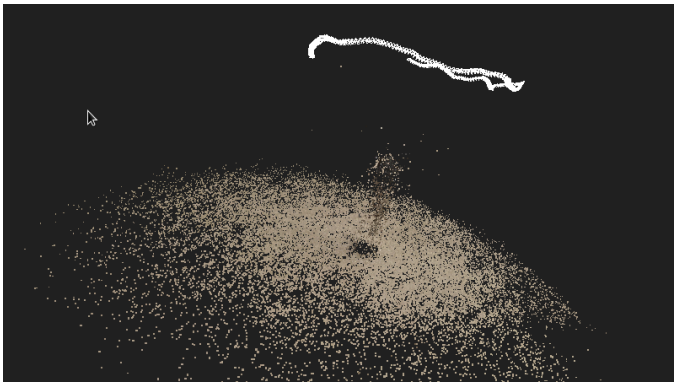

Fig. 2. openSfm 3D Point cloud : PointSize = 0.1

- openSfm 3D Point cloud : PointSize = 0.3

Fig. 3. openSfm 3D Point cloud : PointSize = 0.3

- openSfm 3D Point cloud : PointSize = 0.5

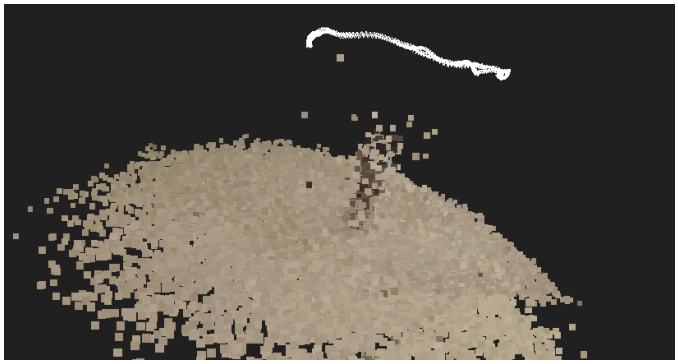

Fig. 4. openSfm 3D Point cloud : PointSize = 0.5

C. Using Agisoft Metashape:

1. Setup and Installation:

Agisoft Metashape is a commercial software for photogrammetry. To set up:

- Download and install the software from the Agisoft official website.
- Obtain a license for full functionality or use the trial version for basic workflows.
- Ensure your system meets the hardware requirements for smooth operation.

2. 3D Point Cloud Generation:

- Import the captured 2D images into the Metashape workspace.
- Align the photos to compute camera positions and generate a sparse point cloud.
- Optimize the alignment and optionally generate a dense point cloud using the reconstruction tools.

3. Outputs/Results:

- The output is a dense 3D point cloud, which can be exported in formats like `.ply`, `.obj`, or `.xyz`.
- Use these files for subsequent neural rendering tasks.

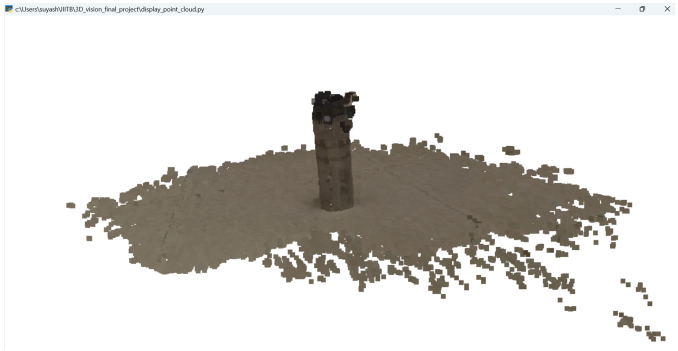

Fig. 5. Agisoft Metashape: 3D Point cloud

OpenSfM Setup with Docker

Step 1: Install Docker

Follow the instructions to install Docker on your system:

Link: https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-16-04

Step 2: Pull OpenSfM Docker Image

Run the following command to download a prebuilt OpenSfM image:

```
sudo docker pull sbstnkll/opensfm
```

Step 3: Run OpenSfM Container

Expose Docker container's port 8000 to localhost:

```
sudo docker run -it --expose 8000 -p 8000:8000
    sbstnkll/opensfm
```

Step 4: Run Reconstruction on Example Images

Navigate to the OpenSfM project directory and run:

```
./bin/opensfm_run_all data/bottle
```

Step 5: Start Python Server

Run the Python HTTP server:

```
python3 -m httpserver
```

Step 6: View 3D Reconstruction in Browser

Copy and paste the following URL into your browser to view the reconstruction:

```
http://localhost:8000/viewer/reconstruction.html#
    file=/data/berlin/reconstruction.meshed.json
```

IV. 3D Rendering Using Gaussian Splatting

Gaussian splatting is an innovative technique for 3D rendering that represents surfaces using a set of overlapping Gaussian kernels in a 3D space. Unlike traditional mesh-based rendering methods, Gaussian splatting leverages continuous volumetric representations, allowing for smooth transitions and realistic visual effects. This approach is particularly effective in handling complex lighting and occlusion scenarios, making it a preferred choice for modern computer graphics applications such as real-time rendering, gaming, and virtual reality.

To facilitate the implementation of Gaussian splatting, **OpenSplat**, an open-source tool, provides a robust framework for rendering and visualization. OpenSplat simplifies the process by offering efficient algorithms and customizable parameters for Gaussian splatting, making it accessible to researchers and developers. With OpenSplat, users can quickly experiment with Gaussian-based 3D rendering techniques and integrate them into their projects.

A. Mathematical Formulation of Gaussian Splatting

Gaussian splatting uses 3D Gaussian kernels to represent points in space. Each Gaussian kernel is defined as:

$$G(\mathbf{x}) = A \cdot \exp\left(-\frac{1}{2}(\mathbf{x}-\mathbf{c})^T \Sigma^{-1} (\mathbf{x}-\mathbf{c})\right)$$

where:

- $\mathbf{x}$ is the 3D coordinate of the point.
- $\mathbf{c}$ is the center of the Gaussian kernel.
- Σ is the covariance matrix, representing the kernel's size and shape.
- A is the amplitude of the Gaussian kernel, controlling its intensity.

In rendering, the Gaussian kernels are projected onto the 2D screen, and their contributions are accumulated to form the final image. The projection can be expressed as:

$$I(u,v) = \sum_{i=1}^{N} G_i(\mathbf{x})$$

where $I(u,v)$ is the intensity at pixel (u,v), N is the number of Gaussians, and $G_i(\mathbf{x})$ is the contribution of the i-th Gaussian kernel.

B. Steps in Gaussian Splatting

Gaussian splatting involves the following key steps:

1) **Point Cloud Input:** A 3D point cloud is used as input, typically generated using photogrammetry tools like OpenSfm or COLMAP.
2) **Gaussian Kernel Initialization:** Each point in the point cloud is associated with a Gaussian kernel. Initial parameters such as position $\mathbf{c}$, size, and intensity A are set based on the input data.
3) **Projection and Accumulation:** Gaussian kernels are projected onto the image plane. Their contributions are accumulated to generate a smooth and continuous representation of the scene.
4) **Optimization:** The parameters of the Gaussian kernels are iteratively optimized to minimize the rendering loss, which is typically defined as the difference between the rendered image and reference images.
5) **Refinement:** The Gaussians are split, duplicated, or pruned based on thresholds for size, gradient, and screen-space coverage. This step ensures efficient representation and improved quality.
6) **Validation and Output:** The final optimized Gaussians are saved in a suitable format (e.g., `.ply` file). Validation can be performed by visualizing the results using tools like **PlayCanvas Viewer**.

C. Advantages of Gaussian Splatting

- Smooth and continuous representation of surfaces.
- Handles complex lighting and occlusion scenarios effectively.
- Flexible and scalable, supporting real-time rendering and applications like virtual reality and gaming.

D. *OpenSplat: Input Requirements*

OpenSplat takes the following input files, typically generated by **OpenSfm**, from the data directory:

- `/images`: A directory containing the input images.
- `reconstruction.json`: A JSON file describing the reconstruction data.
- `image_list.txt`: A text file listing the image paths used in the reconstruction.

These files must be properly prepared and placed in the specified data directory before running OpenSplat.

E. *OpenSplat: Output*

After processing, OpenSplat generates the following output:

- `open_splat.ply`: A PLY file containing the Gaussian splatting 3D point cloud.

The output PLY file can be visualized using the **PlayCanvas Viewer** tool. To view the results:

- Navigate to the PlayCanvas Viewer: https://playcanvas.com/viewer.
- Upload the `open_splat.ply` file.
- Use the zoom-in functionality to closely inspect the reconstructed object.

F. *OpenSplat Experiment-1: Number of iterations*

- **-n, –num-iters:**
  - Defines the number of iterations for the rendering process.
  - Lower values can be used for testing purposes, while higher values are recommended for better quality output.
  - **Default:** 30000.
- OpenSplat 3D Rendering : Number of Iterations = 100

```
./opensplat ~/Downloads/OpenSfM/data/bottle
    -n 100
```

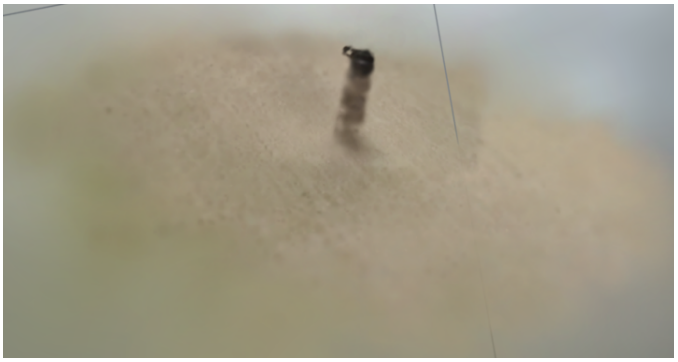

Fig. 6. OpenSplat 3D Rendering : Number of Iterations = 100

- OpenSplat 3D Rendering : Number of Iterations = 500

```
./opensplat ~/Downloads/OpenSfM/data/bottle
    -n 500
```

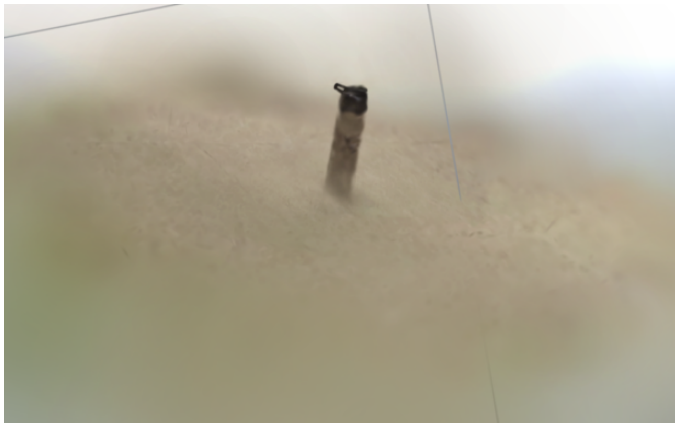

Fig. 7. OpenSplat 3D Rendering : Number of Iterations = 500

- OpenSplat 3D Rendering : Number of Iterations = 1000

```
./opensplat ~/Downloads/OpenSfM/data/bottle
    -n 1000
```

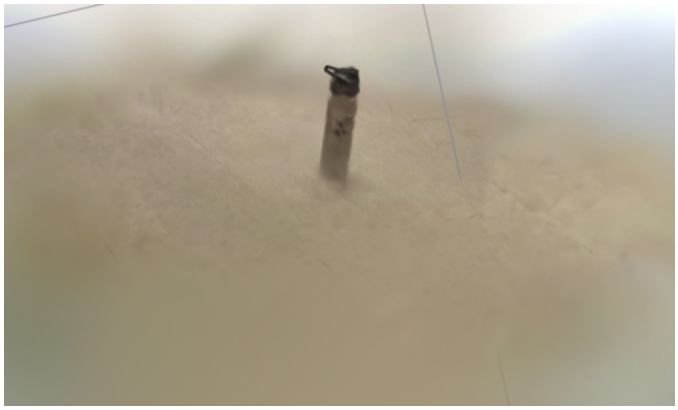

Fig. 8. OpenSplat 3D Rendering : Number of Iterations = 1000

G. *OpenSplat Experiment-2: Downscale factor*

- **-d, –downscale-factor:**
  - Scales input images by a specific factor to reduce resolution.
  - Useful for quick testing or working with limited resources.
  - **Default:** 1.
- OpenSplat 3D Rendering : Downscale factor = 2

```
./opensplat ~/Downloads/OpenSfM/data/bottle
    -n 100 -d 2
```

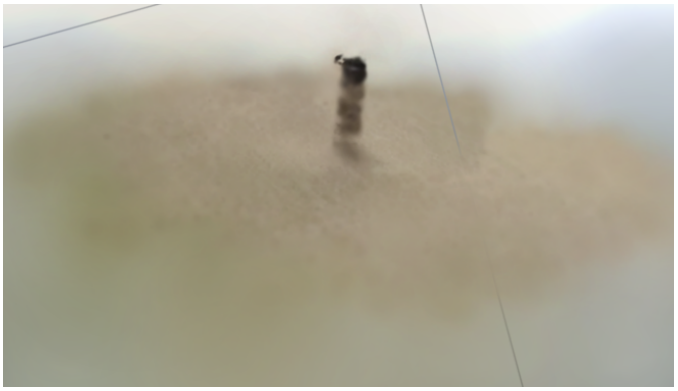

Fig. 9. OpenSplat 3D Rendering : Downscale factor = 2

H. *OpenSplat Experiment-3: Spherical harmonics degree*

- **–sh-degree:**
  - Controls the maximum spherical harmonics degree for lighting effects.
  - Higher degrees produce better quality results but increase computational costs.
  - **Default:** 3.
- OpenSplat 3D Rendering : spherical harmonics degree = 5 and Number of iterations = 100

```
./opensplat ~/Downloads/OpenSfM/data/bottle
    -n 100 --sh-degree 5
```

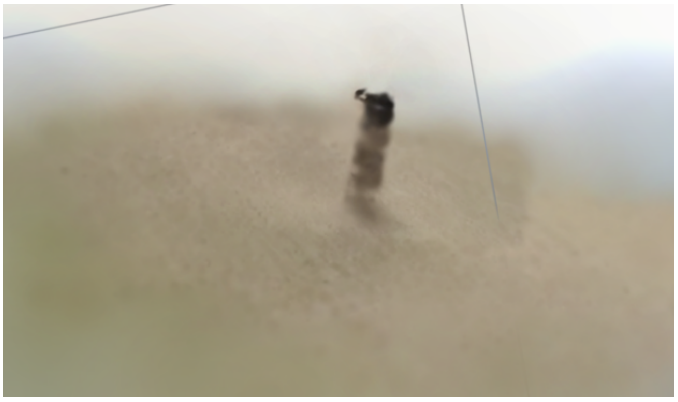

Fig. 10. OpenSplat 3D Rendering : spherical harmonics degree = 5 and Number of iterations = 100

- OpenSplat 3D Rendering : spherical harmonics degree = 5 and Number of iterations = 500

```
./opensplat ~/Downloads/OpenSfM/data/bottle
    -n 500 --sh-degree 5
```

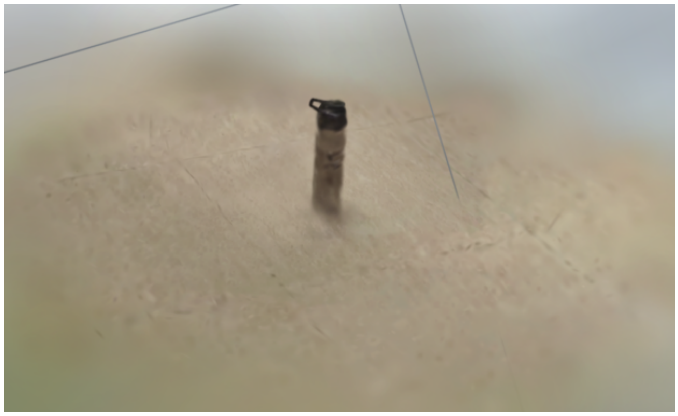

Fig. 11. OpenSplat 3D Rendering : spherical harmonics degree = 5 and Number of iterations = 500

I. *OpenSplat: Observations and Inference*

The observations and inferences derived from experimenting with Gaussian splatting using OpenSplat are summarized below:

- **Number of Iterations:**
  - Increasing the number of iterations enhances the clarity and smoothness of Gaussian splats by allowing finer adjustments during rendering.
  - However, higher iterations significantly increase computational time, making it essential to balance quality with efficiency.
  - **Observation:** For quick experimentation, lower iteration counts (e.g., 1000) provide a reasonable trade-off between speed and output quality. For final renders, higher counts (e.g., 30000) are recommended.
- **Downscale Factor:**
  - Reducing the downscale factor decreases the resolution of input images, leading to faster computations.
  - Lower resolutions, however, result in less detailed Gaussian splats and may omit finer features.
  - **Observation:** A downscale factor of 1 preserves image resolution and provides the best results, though higher factors (e.g., 2 or 4) may be used for testing purposes on resource-constrained setups.
- **Spherical Harmonics Degree:**
  - Increasing the spherical harmonics degree improves lighting accuracy and overall quality by better capturing complex shading and reflections.
  - Higher degrees also demand more computational resources, affecting rendering time.
  - **Observation:** A degree of 5 provides a good balance between visual fidelity and computational cost, while degrees higher than 5 may be reserved for specific scenarios requiring superior lighting effects.
- **Optimal Parameters:**
  - Based on observations, the following parameters are recommended for balanced quality and efficiency:

- ∗ **Number of Iterations:** 1000 (for quick tests), 30000 (for final renders).
- ∗ **Downscale Factor:** 1 (optimal for preserving resolution).
- ∗ **Spherical Harmonics Degree:** 5 (to balance lighting accuracy and computational load).

These observations underscore the importance of parameter tuning in Gaussian splatting to achieve desired outcomes, balancing computational constraints with rendering quality.

OpenSplat Setup

Step 1: Clone the OpenSplat Repository

Run the following command to clone the OpenSplat repository from GitHub:

```
git clone https://github.com/pierotofy/OpenSplat
    OpenSplat
```

Step 2: Navigate to the Project Directory

Move into the OpenSplat directory:

```
cd OpenSplat
```

Step 3: Create a Build Directory

Create a new directory named 'build' and navigate into it:

```
mkdir build && cd build
```

Step 4: Configure and Build OpenSplat

Run the following command to configure and build OpenSplat. Replace '/path/to/libtorch/' with the actual path to your libtorch installation:

```
cmake -DCMAKE_PREFIX_PATH=/path/to/libtorch/ .. &&
    make -j$(nproc)
```

Step 5: Run and Train Gaussian Splatting

To run and train Gaussian splatting on an example dataset, use the following command. Replace ' /Downloads/OpenSfM/-data/bottle' with the path to your data directory:

```
./opensplat ~/Downloads/OpenSfM/data/bottle -n 1000
```

V. 3D Rendering Using NeRF

NeRF, or **Neural Radiance Fields**, is a cutting-edge technique in the field of 3D rendering and scene reconstruction. It uses deep learning to model the volumetric density and color of a 3D scene, enabling the generation of high-quality images from sparse input views. By leveraging neural networks, NeRF can accurately interpolate viewpoints, creating photorealistic renderings and detailed representations of complex geometries. This approach has revolutionized applications such as virtual reality, gaming, and visual effects, providing a powerful tool for synthesizing novel views and reconstructing real-world scenes with unprecedented fidelity.

A. *Theoretical Details of NeRF*

NeRF models a 3D scene by parameterizing it as a continuous 5D function:

$$F_\Theta(\mathbf{x}, \mathbf{d}) = (\mathbf{c}, \sigma)$$

where $\mathbf{x} \in \mathbb{R}^3$ represents a 3D location, $\mathbf{d} \in \mathbb{S}^2$ represents the viewing direction, $\mathbf{c} \in \mathbb{R}^3$ is the RGB color, and $\sigma \in \mathbb{R}^+$ is the volumetric density.

The neural network F_Θ is trained using a collection of 2D images with known camera poses. The rendering equation is approximated by volumetric integration along a ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$:

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

where $T(t) = \exp\left(-\int_{t_n}^{t} \sigma(\mathbf{r}(s))\, ds\right)$ is the transmittance.

This formulation allows NeRF to compute color contributions from multiple sampled points along each ray, resulting in high-quality reconstructions of scenes with complex lighting and geometry.

B. *Results/Outputs:*

Fig. 12. NeRF: Local Model Training

Fig. 13. NeRF: Online Tool Image 1

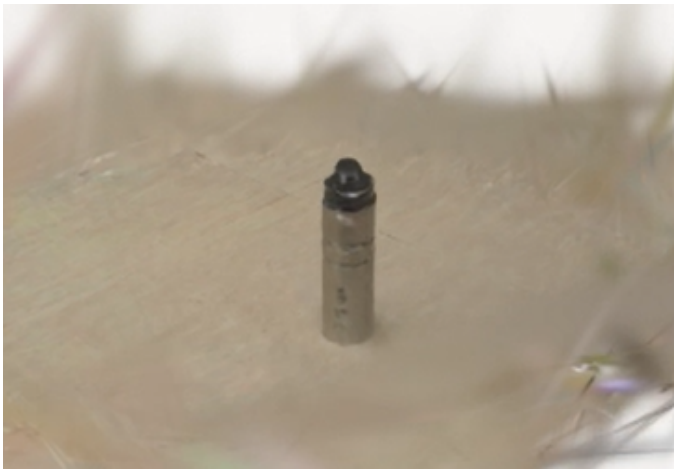

Fig. 14. NeRF: Online Tool Image 2

VI. Comparison between NeRF and Gaussian Splatting

NeRF (Neural Radiance Fields) and Gaussian Splatting are two prominent techniques for 3D rendering, each with its advantages and limitations. While NeRF focuses on volumetric rendering through neural networks, Gaussian Splatting uses a collection of Gaussian kernels to achieve faster and more flexible rendering. To analyze their performance, a detailed comparison is provided based on various metrics such as computational time, rendering quality, memory usage, ease of parameter tuning, and real-time applicability. The results of five experiments conducted for Gaussian Splatting are also included for better insights.

A. NeRF

- **Computational Time:** 3.5 hours.
- **3D Rendering Quality:** Poor than Gaussian Splatting (Need more training)
- **Memory Usage:** More than Gaussian Splatting (Cude required)
- **Ease of Parameter Tuning:** Difficult than Gaussian Splatting (More time needed)
- **Use for Real-Time Applications:** No (Due to more training time)

B. Gaussian Splatting - Experiment 1 : Number of Iterations = 100

- **Computational Time:** 3 mins
- **3D Rendering Quality:** Poor
- **Memory Usage:** 1 GB
- **Ease of Parameter Tuning:** Easy
- **Use for Real-Time Applications:** Yes (faster training)

C. Gaussian Splatting - Experiment 2 : Number of Iterations = 500

- **Computational Time:** 9 mins
- **3D Rendering Quality:** Better than Gaussian Splatting - Experiment 1
- **Memory Usage:** 1 GB
- **Ease of Parameter Tuning:** Easy
- **Use for Real-Time Applications:** Yes (faster training)

D. Gaussian Splatting - Experiment 3 : Number of Iterations = 1000

- **Computational Time:** 21 mins
- **3D Rendering Quality:** Best
- **Memory Usage:** 1.1 GB
- **Ease of Parameter Tuning:** Easy
- **Use for Real-Time Applications:** Yes (faster training)

E. Gaussian Splatting - Experiment 4 : Downscale factor = 2 and Number of Iterations = 100

- **Computational Time:** 2 mins (less than Gaussian Splatting - Experiment 1)
- **3D Rendering Quality:** Poor than Gaussian Splatting - Experiment 1
- **Memory Usage:** 0.9 GB
- **Ease of Parameter Tuning:** Easy
- **Use for Real-Time Applications:** Yes (faster training)

F. Gaussian Splatting - Experiment 5 : Spherical harmonics degree = 5 and Number of Iterations = 100

- **Computational Time:** 18 mins (greater than Gaussian Splatting - Experiment 1)
- **3D Rendering Quality:** Better than Gaussian Splatting - Experiment 1
- **Memory Usage:** 1.1 GB
- **Ease of Parameter Tuning:** Easy
- **Use for Real-Time Applications:** Yes (faster training)

G. *Gaussian Splatting - Experiment 5 : Spherical harmonics degree = 5 and Number of Iterations = 500*

- **Computational Time:** 40 mins (greater than Gaussian Splatting - Experiment 2)
- **3D Rendering Quality:** Better than Gaussian Splatting - Experiment 2
- **Memory Usage:** 1.1 GB
- **Ease of Parameter Tuning:** Easy
- **Use for Real-Time Applications:** Yes (faster training)

VII. Conclusion

In this exploration of 3D rendering techniques, we delved into three core processes that collectively enable the creation of detailed and photorealistic visual representations.

The first step, **3D Point Cloud Generation**, serves as the foundation, transforming 2D images into a spatially aware 3D representation. Tools like **OpenSfm**, **COLMAP** and **Agisoft Metashape** were utilized to extract feature points and generate point clouds, providing a structured basis for further rendering tasks.

Next, **3D Rendering Using NeRF** demonstrated the power of neural networks in synthesizing realistic views from sparse input images. By leveraging Neural Radiance Fields, this method excels in accurately modeling volumetric densities and lighting effects, producing high-fidelity renderings for applications in virtual reality and gaming.

Finally, **3D Rendering Using Gaussian Splatting** introduced an innovative approach that utilizes Gaussian kernels to represent surfaces. This method offers smooth transitions and realistic lighting effects, making it particularly effective in handling occlusions and complex geometries. **OpenSplat** provided a robust framework to experiment with this technique, enabling efficient and customizable rendering workflows.

Together, these methods represent a comprehensive pipeline for 3D reconstruction and rendering, combining the precision of point cloud generation, the neural interpolation capabilities of NeRF, and the smooth volumetric representations of Gaussian splatting. This integration paves the way for advancements in computer graphics, immersive technologies, and real-time rendering applications.

References

[1] OpenSfM Tutorial, Mapillary, 2024. [Online]. Available: https://github.com/vidmap/opensfm-tutorial/blob/master/README.md. [Accessed: Dec. 17, 2024].

[2] OpenSplat, Piero Toffanin, 2024. [Online]. Available: https://github.com/pierotofy/OpenSplat. [Accessed: Dec. 17, 2024].

[3] Instant NeRF: Instant Neural Graphics Primitives, NVIDIA, 2024. [Online]. Available: https://github.com/NVlabs/instant-ngp. [Accessed: Dec. 17, 2024].

[4] Colmap - Structure-from-Motion and Multi-View Stereo, Johannes L. Schönberger, 2024. [Online]. Available: https://colmap.github.io/. [Accessed: Dec. 17, 2024].

[5] Agisoft Metashape, Agisoft LLC, 2024. [Online]. Available: https://www.agisoft.com/. [Accessed: Dec. 17, 2024].