ROSplat: A Plug-and-Play Visualizer for 3D Gaussian Splatting in ROS2

Shady Gmira

Abstract

We introduce ROSplat, the first real-time visualizer for 3D Gaussian Splatting fully integrated with the ROS2 ecosystem. ROSplat can visualize 3D Gaussian rendering, IMU data, and image streams via native ROS2 communication at real-time speed. Designed for scalability and ease of use, ROSplat supports adaptive buffer updates and different sorting backends, allowing smooth rendering performance even with millions of splats. Being tailored specifically for high-performance and incremental map update, ROSplat simplifies development and debugging in 3D Gaussian-based odometry, mapping, and other vision-based robotics tasks.

I. Introduction

3D Gaussian Splatting (3DGS) is an emerging rendering technique that represents scenes using a collection of Gaussian primitives. This approach enables photorealistic scene reconstruction with high computational efficiency, making it particularly attractive for real-time applications. SLAM systems based on 3DGS have demonstrated advantages such as enhanced visual fidelity, robust mapping, and improved computational performance [1], [2].

Despite these benefits, current 3DGS-based SLAM implementations typically require the development of dedicated visualization tools to monitor reconstruction progress. This additional requirement can hinder the adoption and streamlined deployment of such systems. Therefore, we propose ROSplat, a ROS-enabled visualizer for 3D Gaussian Splatting, designed to integrate seamlessly with existing robotics middleware and simplify the visualization process. Additionally, ROSplat supports the visualization of IMU and image data, providing a more complete view of system dynamics. This tool facilitates debugging, development, and demonstration of 3DGS-based methods within ROS-based robotic platforms.

II. Methodology

A. System Architecture

The ROSplat framework is architected as a collection of interdependent modules that work together to achieve real-time visualization:

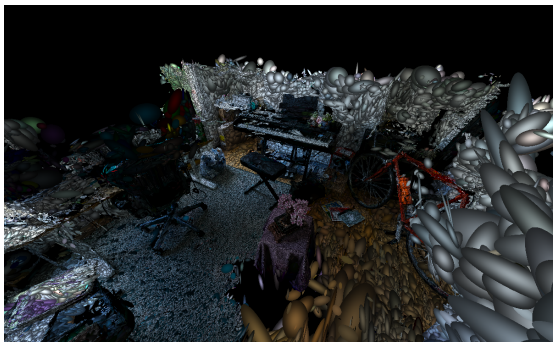
- **Data Processing Module:** At the core is the `GaussianData` class, which encapsulates arrays for positions (`xyz`), rotations (`rot`), scales (`scale`), opacities (`opacity`), and spherical harmonics (`sh`). The method `flat()` converts these structured parameters into a contiguous array so that they can be transfered to the GPU.
- **Rendering Module:** OpenGL is used to render Gaussian data on the screen. The `OpenGLRenderer` class compiles shader programs, manages vertex array objects (VAOs) and shader storage buffer objects (SSBOs), and handles the sorting of Gaussian elements based on depth. Buffer management is optimized via conditional updates—full or partial—depending on data changes.
- **User Interface Module:** The framework uses the ImGuiBundle [3] library to provide an interactive GUI. Users can manipulate camera views, adjust rendering parameters, and view the changes in real-time updates.

B. Data Processing and Gaussian Management

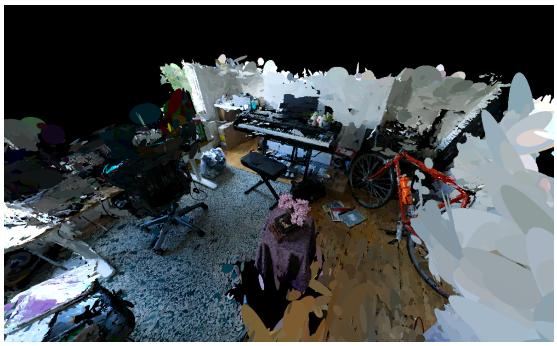
In ROSplat, each Gaussian element is parameterized to represent its spatial, geometric, and visual characteristics. These include its position in three-dimensional space, orientation represented as a quaternion, anisotropic scaling along each axis, an opacity term controlling transparency, and a set of spherical harmonics coefficients that capture low-order lighting information. This description of Gaussians is identical to the one presented by Kerbl et al. [4].

A new ROS message type was created that contains all of the aforementioned parameters. Individual Gaussian primitives are represented by a dedicated message type, while collections of Gaussians are grouped into aggregate messages for batch processing and visualization. In addition to streaming Gaussians from ROS topics, ROSplat also supports offline visualization. Users can directly load existing 3D Gaussian Splatting reconstructions from PLY files through the graphical interface.

The authors are with the Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence, University of Groningen, Groningen, The Netherlands. E-mails: `s.gmira@student.rug.nl`

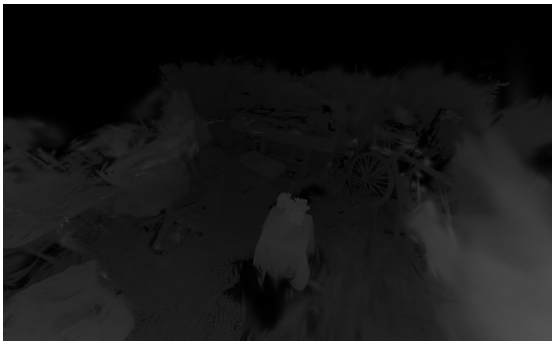


(a) Gaussian Ball



(b) Flat Ball

(c) Billboard



(d) Depth

Fig. 1: Visualization modes in ROSplat.

C. Rendering Pipeline and GPU Optimization

The rendering module in ROSplat is built on OpenGL and is designed to support real-time visualization of dynamically updated 3D Gaussians. The implementation draws inspiration from the *GaussianSplattingViewer* project [5], which uses OpenGL to render Gaussian primitives with high performance and visual fidelity. One drawback from *GaussianSplattingViewer* is that it is only able to visualize static scene. On the other hand, ROSplat is tailored to accommodate dynamic updates typical of SLAM systems, where the scene evolves incrementally over time.

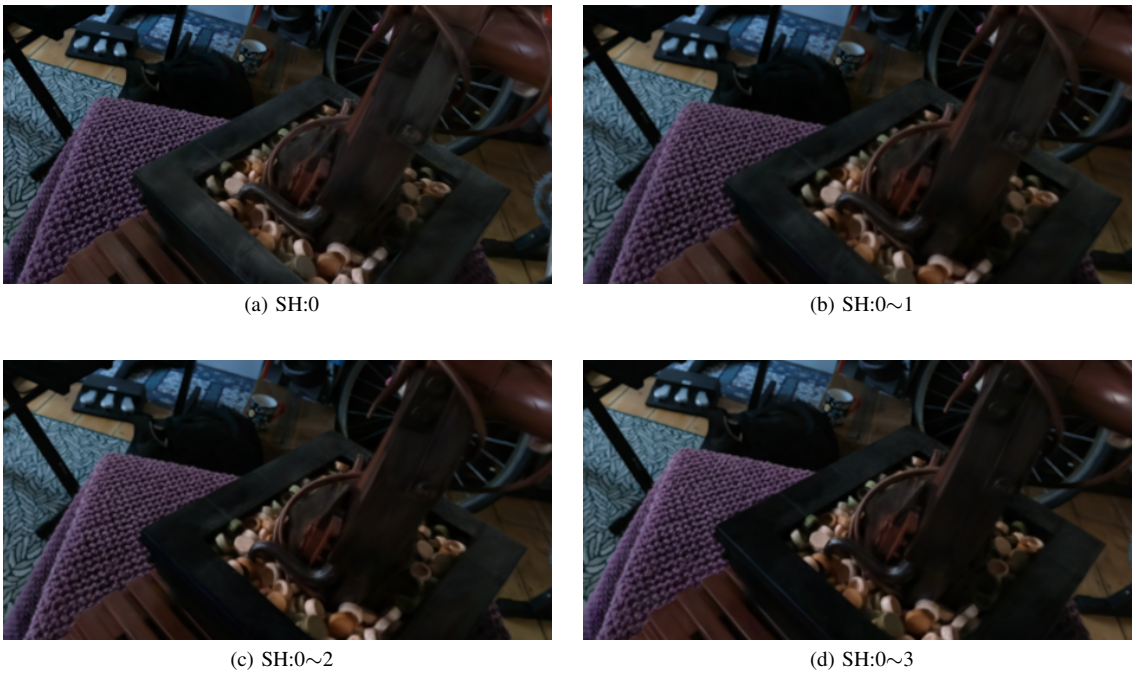
The renderer uses custom vertex and fragment shaders that process and visualize Gaussian primitives on the GPU. Each Gaussian is defined by parameters such as position, orientation, scale, opacity, and spherical harmonics coefficients. The vertex shader transforms each Gaussian into screen space using the current view-projection matrix, while the fragment shader computes color contributions of each Gaussian based on opacity and shading using low-order spherical harmonics.

Gaussian parameters are stored and transmitted to the GPU using Shader Storage Buffer Objects (SSBOs), which allow large and flexible datasets to be accessed directly in shaders. To optimize memory transfer and maintain real-time performance, ROSplat implements an adaptive buffer update strategy. Whenever Gaussians are sent to a ROS node, the system checks the number of active Gaussian entries and triggers either a full buffer reallocation or a partial update. Partial updates are used when only a subset of splats changes between frames, which is common in SLAM systems where the map is incrementally updated. In contrast, full updates are infrequent and are primarily used to clear the existing SSBO of outdated data. As more Gaussians are added, the SSBO size increases, potentially leading to VRAM exhaustion. The use of partial updates mitigates redundant memory transfers and allocation which is important to maintain real-time rendering performance.

As part of the Gaussian Splatting pipeline, it is necessary to order the splats by depth with respect to the current camera pose. ROSplat supports multiple sorting backends for this purpose, including CPU-based sorting and GPU-accelerated methods implemented using CuPy and PyTorch. Sorting is performed every frame based on the updated view matrix to ensure correct blending of overlapping splats. The sorted indices are then passed to the renderer for correct rendering order during draw calls. A diagram of the full system architecture, is provided in Figure 7 for reference.

D. User Interface and Interaction

The layout of the user interface is organized into multiple tabs, each dedicated to a specific aspect of the system. The ROS Settings tab allows users to configure the ROS topics to which the visualizer subscribes. Currently, the system automatically detects and displays all available topics; however, it supports only a limited set of message types, specifically RGB images, IMU data, and messages of type `GaussianArray`.



(a) SH:0 (b) SH:0∼1

(c) SH:0∼2 (d) SH:0∼3

Fig. 2: Visualization modes in ROSplat using different orders of spherical harmonics.

The 3D Visualization tab provides a real-time rendering view that shows how the virtual camera (associated with the visualizer) moves through 3D space. This feature offers immediate visual feedback about the scene being reconstructed and is particularly useful for understanding camera trajectories and viewpoint changes. Although the current implementation focuses on visualizing the camera path, the system is designed with extensibility in mind and can be expanded to support additional pose-related visualizations, such as real-time ground-truth or estimated poses from external tracking systems.

Complementing the 3D view, the Frames tab displays incoming RGB image frames, which serve as an important point of reference for interpreting what the SLAM system is seeing. The IMU Data tab presents accelerometer and gyroscope measurements over time. Together, these interface components offer essential debugging capabilities for the development and evaluation of SLAM algorithms, allowing a user to inspect both high-level system behavior and low-level sensor data in an integrated environment.

E. Visualization Modes and Shader Control

ROSplat offers users control over how Gaussian splats are visualized, allowing users to switch between multiple rendering modes at runtime. This feature is accessible via the GUI.

Specifically, users can select from a range of visualization types using a drop-down menu in the “ROSPlat” tab, as illustrated in Figure 1. In addition, users can toggle different spherical harmonics (SH) components to facilitate debugging, as shown in Figure 2.

This system is implemented through a runtime switch that updates the renderer’s active shader. The selected mode controls how each Gaussian’s attributes are interpreted and visualized on screen. This modular shader architecture enables ROSplat to support multiple rendering strategies without recompiling the application or restarting the GUI.

III. Results

All experiments were executed on a single machine equipped with an NVIDIA RTX 3070 Mobile GPU (115W, 8GB VRAM), a 16-core AMD Ryzen 7 5800H CPU, and 32GB of RAM. Each test was repeated three times using the same dataset, “kitchen”, which is an official pre-trained scene provided in PLY format by the authors of 3D Gaussian Splatting and hosted on the INRIA repository[1] [**?**]. A custom script was used to extract and publish subsets of the data in a controlled and reproducible manner. Specifically, the script iteratively loads 1000 Gaussians at a time and publishes them as an individual `GaussianArray` message, which emulates the behavior of a SLAM system. Although the semantic content of the scene is not critical for this evaluation, the number and structure of Gaussians remain consistent across all runs. This uniformity ensures a fair comparison of sorting and rendering performance under varying backend configurations.

[1] https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/datasets/pretrained/models.zip

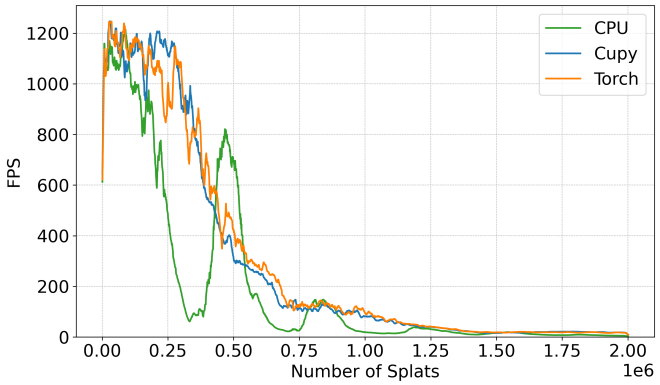


Fig. 3: Frames Per Second (FPS) vs. number of splats for different sorting backends.

To evaluate the performance of different sorting backends used in the rendering pipeline, we benchmarked the system by measuring the frame rate (FPS) as a function of the number of 3D Gaussians rendered. The results, visualized in Figure 3, demonstrate the relative efficiency of GPU-accelerated sorting methods—Torch and CuPy—compared to the CPU-based baseline. As shown in the plot, with a small quantity of 3D Gaussians (less than 100,000), all three backends achieves more than 1000 FPS, with the CPU backend being slightly slower. However, starting from roughly 100,000 splats, the CPU backend exhibits a rapid decrease in FPS, and eventually fails to render in real time. The performance drop of Torch happens at around 250,000 Gaussians, while the Cupy backend starts to slow down with 300,000 Gaussians. The curves of Torch and Cupy are comparable to each other, and their performance degradation happens much later than the CPU backend.

To further quantify the performance differences between sorting backends, we computed the average relative FPS improvements across trials. As shown in Figure 4, Torch provides a 23.2% increase in frame rate compared to the CPU baseline, while CuPy achieves a 21.9% improvement.

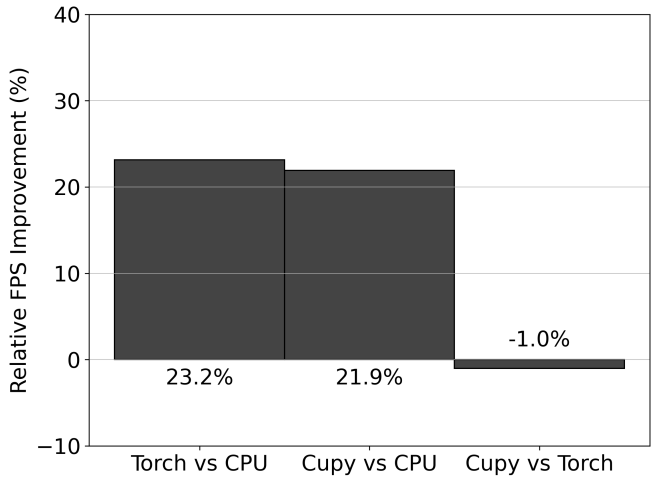


Fig. 4: Relative FPS improvement (%) averaged across trials for each backend comparison.

These findings reinforce the importance of GPU-accelerated sorting in real-time systems. The CPU backend is unable to meet the computational demands of rendering a large number of Gaussians, while the CuPy and Torch implementations offer sustained interactive performance even under high load.

In addition to performance, we examined memory usage on both the CPU and GPU to understand the scalability of the system. Figures 5 and 6 illustrate average RAM and VRAM usage, respectively, as the number of active Gaussians increases.

RAM usage, shown in Figure 5, grows steadily and smoothly with the number of splats. This trend is expected since the system stores all Gaussians in memory, including parameters such as position, rotation, scale, and spherical harmonics. The continuous increase in RAM reflects linear growth in data storage requirements on the host side, which scales with the total number of splats.

VRAM usage, shown in Figure 6, exhibits a different pattern. Rather than a smooth increase, it features noticeable spikes followed by plateaus. This behavior is a direct result of the buffer allocation strategy implemented in the system. Specifically,

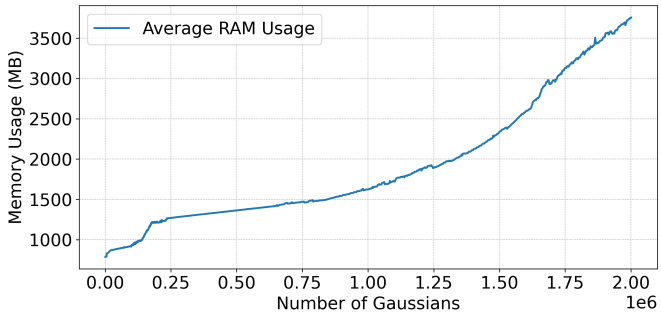


Fig. 5: Average RAM usage as a function of the number of Gaussian splats.

the Shader Storage Buffer Object (SSBO) used to store Gaussian data on the GPU is dynamically resized: whenever the number of splats exceeds the current buffer capacity, the buffer is reallocated with double the previous size. These doubling events lead to sudden increases in VRAM usage (the numbers are not exactly doubled since PyTorch allocates a fixed amount of memory for internal usage), followed by stable regions where the existing buffer is sufficient to hold new splats. This approach minimizes the number of reallocations and improves runtime performance, but it results in a staircase-like memory usage curve.

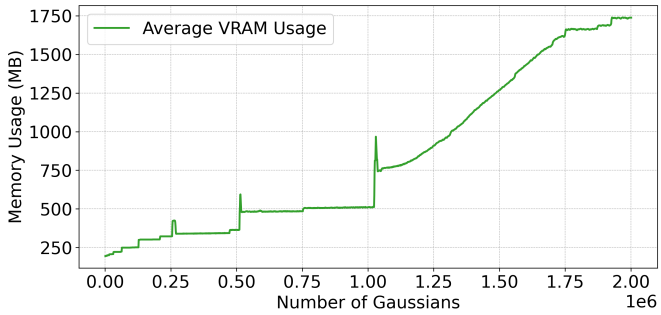


Fig. 6: Average VRAM usage as a function of the number of Gaussian splats.

Interestingly, both the VRAM and RAM usage curves show a noticeable change in growth pattern after approximately one million Gaussians. This shift reflects distinct behaviors in how memory is managed on the GPU and CPU, respectively.

On the GPU side, the VRAM usage demonstrates a smooth increase after this threshold. This is explained by how Shader Storage Buffer Objects (SSBOs) are allocated and reported. Allocating a buffer on the GPU does not immediately consume memory; usage only increases as the buffer is filled with data. In the early stages, when the buffer is small, frequent reallocations occur to accommodate the growing number of splats. Once the buffer becomes large enough, reallocations are needed less often, and new data fills the pre-allocated space. Because most GPU drivers report only used rather than allocated memory, this results in a smoother and more linear growth pattern that eventually flattens as the buffer stabilizes.

It can be seen that the VRAM usage for two million Gaussians is more than three times higher than one million Gaussians. This is because each frame, the system creates a temporary buffer for the 3D positions of the Gaussians via the appropriate Torch and CuPy function calls. These operations upload data from the CPU memory to the GPU memory. Although the buffer is overwritten frame-to-frame, PyTorch and CuPy manage these tensors using internal memory pools with different caching strategies. Rather than freeing the memory immediately, they often keep allocations for potential reuse, especially for larger tensors. This behavior can delay garbage collection and lead to VRAM usage growth, especially if the allocator overestimates future reuse or if the object size falls into a caching threshold.

On the CPU side, the RAM usage begins to deviate from the earlier linear trend, showing signs of near-quadratic growth after one million Gaussians. This is due to internal memory duplication during sorting operations. Specifically, one persistent list of Gaussians is maintained by the WorldSettings object, while an additional temporary copy, containing only the xyz positions, is created for sorting in each frame. While this duplication is limited in scope, it becomes increasingly significant as the dataset size grows, contributing to the observed acceleration in RAM consumption beyond the one-million-splat mark.

Together, these evaluations confirm that the proposed system supports scalable, high-performance rendering of large Gaussian splat sets with efficient memory management on both CPU and GPU. The adaptive buffer update and sorting strategies contribute significantly to maintaining interactivity and responsiveness under heavy load.

IV. Conclusion

We introduced ROSplat, a real-time visualization tool that connects 3D Gaussian Splatting with the ROS2 ecosystem. By combining GPU-accelerated rendering with native ROS2 messaging, ROSplat allows fast and scalable visualization of large-scale 3D scenes, which is essential for tasks like SLAM and scene reconstruction.

ROSplat is easy to set up and use, supports live visual feedback, and provides useful tools for debugging and development. It handles millions of splats efficiently and supports additional sensor data like images and IMU readings. Our experiments show that the system performs well even in demanding settings, making it a practical tool for robotics research.

In the future, ROSplat could be extended with better user controls, support for more data types, and integration with physics simulators such as MuJoCo or Gazebo. This would make it possible to test and train vision-based systems in simulation with photorealistic visual feedbacks. We see ROSplat as a step toward making advanced rendering more accessible and useful for the robotics community.

Acknowledgments

I'm glad to have worked on such a challenging topic and grateful for the invaluable advice and support I received throughout this project. Special thanks to Qihao Yuan and Kailai Li for their guidance and encouragement as well as the constructive feedback that helped shape this work.

This project was additionally influenced by limacv's implementation of the GaussianSplattingViewer repository.

References

[1] H. Matsuki, R. Murai, P. H. J. Kelly, and A. J. Davison, "Gaussian Splatting Slam," in *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024, pp. 18 039–18 048. [Online]. Available: https://doi.org/10.1109/CVPR52733.2024.01708

[2] L. C. Sun, N. P. Bhatt, J. C. Liu, Z. Fan, Z. Wang, T. E. Humphreys, and U. Topcu, "MM3DGS SLAM: Multi-modal 3D Gaussian Splatting for SLAM Using Vision, Depth, and Inertial Measurements," in *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2024, pp. 10 159–10 166. [Online]. Available: https://doi.org/10.1109/IROS58592.2024.10802389

[3] pthom, "imgui_bundle," https://github.com/pthom/imgui_bundle.

[4] B. Kerbl, G. Kopanas, T. Leimkuehler, and G. Drettakis, "3d gaussian splatting for real-time radiance field rendering," *ACM Trans. Graph.*, vol. 42, no. 4, Jul. 2023. [Online]. Available: https://doi.org/10.1145/3592433

[5] LimaCV, "Gaussiansplattingviewer," https://github.com/limacv/GaussianSplattingViewer, 2023.

APPENDIX A

Figure 7 presents an overview of the ROSplat system architecture, the major components and their interactions. The system is organized into three primary blocks, each representing a step in the data flow: system inputs, visualization and rendering, and user interaction.

The System Inputs section (green block) includes both static and dynamic data sources. On the left, the PLY Loader reads pre-trained Gaussian splat data from disk, which can be used independently of the ROS interface. On the right, the ROS Interface manages live ROS2 topics, including a Gaussian ROS node and additional nodes that stream image and IMU data. This setup allows ROSplat to support both offline and real-time visualization use cases.

The Visualization and Rendering Core (purple block) forms the heart of the system. Gaussian data, whether loaded from file or received from ROS, is passed to a GPU data manager. This module handles Shader Storage Buffer Objects (SSBOs), sorting of splats per frame, and deciding between partial/full memory updates. The OpenGL Shader Pipeline then renders the Gaussian splats using these processed buffers. At the same time, sensor data from ROS topics is displayed in dedicated visualization tabs, allowing users to monitor image streams and IMU outputs.

Finally, the User Interface (bottom block) allows user interaction through an intuitive graphical frontend. Users can navigate the scene using keyboard controls (e.g., WASD), modify rendering settings such as visualization mode, enable/disable the sorting of Gaussians, and load PLY files via the GUI. All user inputs affect the rendering pipeline in real time.

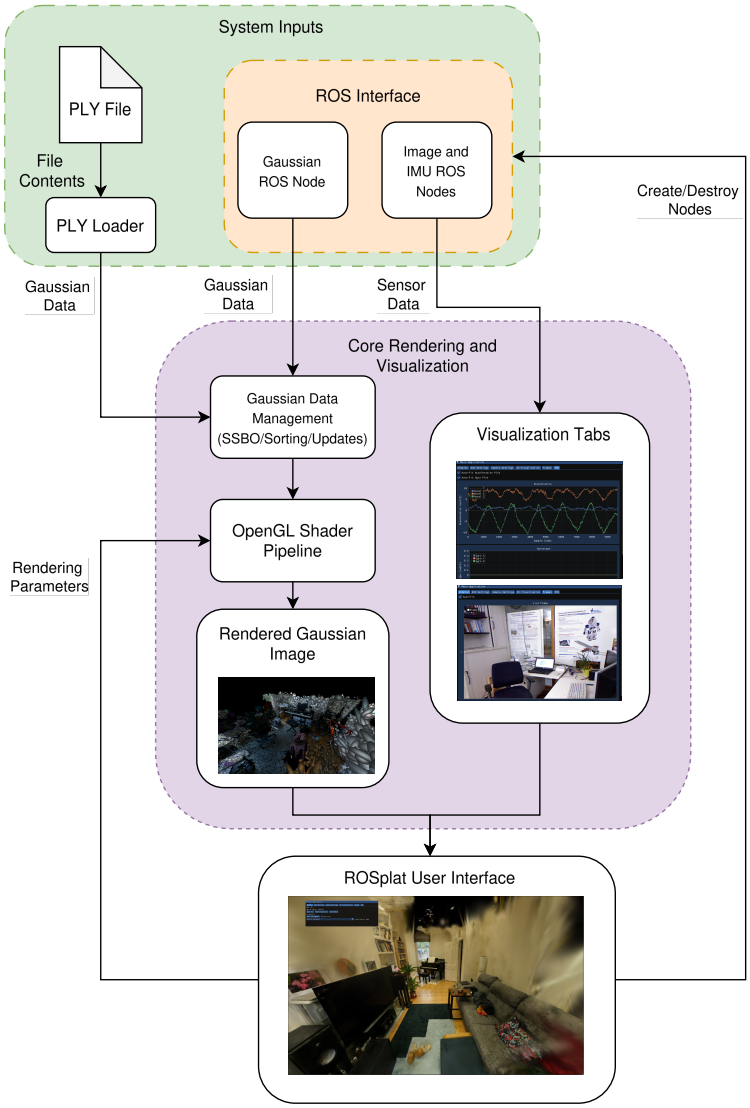


Fig. 7: System architecture of ROSplat showing the flow of data from system inputs (PLY files and ROS messages) through the rendering and visualization components to the user interface.