3dgs demo

tavasoli

October 2025

1 Overview of the 3D Gaussian Splatting (3DGS) Implementation

Purpose. This code implements a minimal, self-contained version of *3D Gaussian Splatting (3DGS)* in PyTorch. It renders a small set of colored, semi-transparent 3D Gaussian blobs into a 2D image using a pinhole camera model. The entire pipeline is differentiable, allowing optimization of blob parameters such as position, color, and opacity via gradient descent.

1.1 Main Components

Camera and Geometry.

- `look_at(eye, at, up)` constructs a right-handed world-to-camera view matrix. Points in front of the camera have positive depth $(+Z)$ in camera coordinates.
- `project_points(Xc, K)` applies the pinhole camera projection to obtain pixel coordinates (u, v) and depth z.
- `jacobian_proj(Xc, K)` computes the 2×3 Jacobian of the projection function. This linearizes the mapping to approximate the 2D screen-space covariance of each Gaussian.

Rendering.

- `render_gaussians(...)` is the core differentiable renderer:
  1. Transforms each Gaussian mean from world space to camera space.
  2. Projects the 3D mean to 2D pixels.
  3. Uses the projection Jacobian to map the isotropic 3D variance σ^2 to a 2D elliptical footprint.
  4. Performs front-to-back compositing (near-to-far) using soft α blending over a white background.
  5. Returns a rendered RGB image of size $H \times W$ in $[0, 1]$.

1.2 Demonstration Functions

`demo()`. Generates eight random 3D Gaussians in front of the camera and renders a 256×256 image. The output is a collection of smooth, colored, semi-transparent blobs.

`tiny_optimize_demo()`. Creates a ground-truth image from three fixed Gaussians, then optimizes a randomly initialized scene to reconstruct the target image by minimizing the mean-squared error (MSE) loss using the Adam optimizer. This demonstrates end-to-end differentiability of the renderer.

1.3 Sanity Tests

The function `_quick_tests()` performs simple automatic tests to verify correctness:

- A black Gaussian blob darkens the white background at the center pixel.
- A nearer red Gaussian dominates over a farther blue Gaussian when they overlap.
- Points located behind the camera (negative camera-space depth) are correctly culled.

1.4 Conceptual Summary

Each 3D point is represented as an isotropic Gaussian density with color and opacity. The projection is linearized to obtain a 2D Gaussian footprint in screen space. Front-to-back α compositing blends splats smoothly to create the final image. Because all operations are implemented in PyTorch, the rendering process is fully differentiable.

1.5 Limitations and Extensions

- The current version uses only isotropic 3D Gaussians and no acceleration structures, so it is slow for large scenes.
- Possible extensions include anisotropic (rotated) Gaussians, multi-view training, and real-data integration.
- The renderer can be extended to load real datasets such as KITTI or Blender-NeRF scenes for 3D reconstruction tasks.