A Conceptual and Technical Exploration of Gaussian Splatting: Metric Sandwich Operators, Robotics Integration, and a Prototype Python Renderer

Bjørn Remseth
Email: la3lma@gmail.com
With AI assistance

***Abstract*—This document consolidates a wide-ranging technical exploration of Gaussian splatting, its mathematical underpinnings, its relationship to geometric algebraic operators, integration with robotics frameworks (e.g., ROS 2 and MoveIt), dynamic articulated Gaussian models, world-model learning pipelines, and a complete Python implementation of a prototype splat renderer. The discussion moves from foundational derivations (projection Jacobians, spherical harmonic lighting gradients, alpha compositing, covariance pushforwards) to advanced architecture sketches for robotic manipulation and world-model learning. Expansion points are clearly marked for further work. Several illustrative diagrams are included to visualize key concepts and system architecture.**

***Index Terms*—Gaussian splatting, 3D reconstruction, geometric algebra, robotics integration, ROS 2, differentiable rendering, world models**

`Version: 2025-11-17-10:13:12-CET-aac0e2d-main`

Note on document creation: This document was created with AI assistance. See the "About This Document" section for details on the methodology and verification processes used.

I. Introduction

Gaussian splatting has emerged as a high-performance, explicit representation for complex 3D scenes. Unlike implicit methods such as Neural Radiance Fields (NeRFs) [1], Gaussian splats provide:

- Analytic projection to 2D via Jacobians,
- Real-time rendering through rasterized elliptical footprints,
- Differentiable parameters (positions, covariances, colors, SH lighting),
- Extremely fast training compared to NeRF-style volumetric rendering,
- Direct compatibility with robotics pipelines.

A Gaussian splat refers to a 3D Gaussian blob that carries color and opacity, which can be projected onto a 2D image plane [2], [3]. Each Gaussian is defined by parameters describing its position, size/shape, and appearance:

- **Position**: 3D coordinates (X, Y, Z) of the Gaussian's center in space
- **Covariance**: A 3×3 covariance matrix Σ defining how the Gaussian is stretched or scaled in different directions (encoding the ellipsoidal shape)
- **Color**: An RGB value for the Gaussian's color
- **Alpha**: An opacity value α indicating transparency

This document brings together deep mathematical derivations, conceptual connections to geometric algebra, and engineering-oriented designs for integrating Gaussian splats into robotics systems and learned world models.

Finally, a complete Python program demonstrates an end-to-end rendering pipeline for a simple *blue cube*, entirely using Gaussian splats. The implementation is provided in the accompanying `python_renderer/` directory.

II. Mathematical Foundations

A. Why the Gaussian Exponent is a Quadratic Sandwich

A multivariate Gaussian density uses the quadratic form

$$(\mathbf{x} - \mu)^T \Sigma^{-1} (\mathbf{x} - \mu) \tag{1}$$

because it is uniquely determined by deep mathematical constraints:

1) Ellipsoidal level sets: The only function with elliptical level sets and smooth radial falloff is a quadratic form.

2) Maximum entropy distribution: The Gaussian is the maximum entropy distribution subject to fixed mean and covariance. Solving the variational problem yields a quadratic exponent.

3) Mahalanobis distance: The quadratic form defines the Mahalanobis distance induced by the metric tensor Σ^{-1}.

4) Metric interpretation: Σ^{-1} serves as a metric tensor. Large variance $\leftrightarrow$ small precision $\leftrightarrow$ shallow curvature. Small variance $\leftrightarrow$ sharp curvature.

B. Representing 3D Gaussians

Mathematically, a 3D Gaussian can be thought of as a Gaussian function (a 3D normal distribution) centered at a point. The covariance matrix Σ (3×3) associated with a

Gaussian encodes its shape and size. The Gaussian function for a point $\mathbf{x}$ in space is:

$$G(\mathbf{x}) = \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^\top \Sigma^{-1}(\mathbf{x}-\mu)\right), \tag{2}$$

where μ is the center (mean) and Σ is the covariance matrix controlling the spread.

1) Covariance as an Ellipsoid: The covariance matrix Σ of a Gaussian is symmetric and positive semi-definite. Geometrically, Σ defines an ellipsoid: the set of points $\mathbf{x}$ such that $(\mathbf{x}-\mu)^\top\Sigma^{-1}(\mathbf{x}-\mu) = c$ forms an ellipsoidal surface [4]. The orientations and lengths of the ellipsoid's principal axes are given by the eigenvectors and eigenvalues of Σ.

2) Parameterizing via Rotation and Scale: In practice, directly optimizing the 9 entries of Σ while guaranteeing positive semi-definiteness is challenging. The authors of 3D Gaussian Splatting use a clever reparameterization [3], [4]: each Gaussian's shape is defined by a diagonal scaling matrix and a rotation:

$$\Sigma = R\,S\,S^T\,R^T, \tag{3}$$

where S is a diagonal matrix containing three scale factors (s_x, s_y, s_z), and R is a 3×3 rotation matrix (derived from a stored quaternion). This guarantees Σ is always positive semi-definite for any choice of R and S.

III. Projection and Rendering

A. Projection Jacobian: 3D Gaussian to 2D Ellipse

Projection proceeds through:

- World $\to$ camera (R, t),
- Camera $\to$ pixel via pinhole intrinsics (f_x, f_y, c_x, c_y).

The projection Jacobian is:

$$J_{cam} = \begin{pmatrix} \frac{f_x}{z} & 0 & -f_x\frac{x}{z^2} \\ 0 & \frac{f_y}{z} & -f_y\frac{y}{z^2} \end{pmatrix} \tag{4}$$

The projected covariance is:

$$\Sigma_{2D} = J\,\Sigma\,J^T. \tag{5}$$

This ellipse determines the 2D footprint of each Gaussian splat. To render these Gaussians on a 2D image, we must project each 3D Gaussian onto the image plane. This involves projecting the center position to image coordinates (u, v) and projecting the covariance to a 2×2 covariance matrix for the 2D ellipse [2], [4].

If J represents the Jacobian of the projection and W represents the world-to-camera transform, then the projected covariance can be written as:

$$\Sigma_{2D} = J\,W\,\Sigma\,W^T\,J^T. \tag{6}$$

1) Ensuring Numerical Stability: A small tweak is applied to Σ_{2D} to avoid numerical issues. The 2D covariance might be singular or nearly so. To ensure invertibility, a tiny value (e.g., 0.3) is added to each diagonal entry [4]. This makes Σ_{2D} strictly positive-definite.

2) Ellipse Radius Approximation: The authors choose the radius of a circle that would circumscribe the ellipse to simplify rasterization [4]. For a 1D Gaussian, about 99.73% of its mass lies within 3 standard deviations. By analogy, they use 3σ as the cutoff for the footprint.

For the ellipse, the largest eigenvalue $\lambda_{\max}$ of Σ_{2D} corresponds to the longest principal axis. The radius r is chosen as:

$$r = 3\sqrt{\lambda_{\max}}. \tag{7}$$

The eigenvalues of a 2×2 matrix $\Sigma_{2D} = \begin{pmatrix} a & b \\ b & c \end{pmatrix}$ are:

$$\lambda_{1,2} = \frac{a+c\pm\sqrt{(a-c)^2+4b^2}}{2}. \tag{8}$$

B. Spherical Harmonic Lighting Gradients

Per-splat appearance is encoded using SH coefficients

$$C(\mathbf{d}) = \sum_i c_i Y_i(\mathbf{d}). \tag{9}$$

Gradients w.r.t. SH coefficients are analytic:

$$\frac{\partial\mathcal{L}}{\partial c_i} = (C - C^*)\,Y_i(\mathbf{d}). \tag{10}$$

Gradients w.r.t. viewpoint likewise derive from analytic SH derivatives.

C. Front-to-Back Alpha Compositing

For each pixel, with accumulated color $\mathbf{C}$ and transmittance T:

$$\mathbf{C} \leftarrow \mathbf{C} + T\,\alpha_i\,\mathbf{C}_i, \qquad T \leftarrow T(1-\alpha_i). \tag{11}$$

This approximates the continuous volume rendering integral used in NeRF [1] with a discretized formulation suitable for splats. The rasterization approach is designed to be fast (using GPU acceleration) and differentiable [2], enabling gradient-based optimization.

D. Rendering with Millions of Gaussians

The basic procedure for Gaussian rasterization is [2]:

1) Project each Gaussian to get its 2D position and radius
2) Sort Gaussians by depth (distance from the camera)
3) Blend contributions for each pixel front-to-back

Each Gaussian contributes an additive color blended by its alpha (transparency) to the pixels within its radius. By summing up contributions from all Gaussians, we obtain the final pixel colors.

IV. Training and Optimization

A. Obtaining and Refining the Gaussians

The training process combines classical reconstruction with iterative refinement [2]:

1) Initialization from Structure-from-Motion: The process begins with running a Structure from Motion (SfM) algorithm on input photographs. SfM produces a sparse point cloud – a set of 3D points with colors that correspond to features in the images.

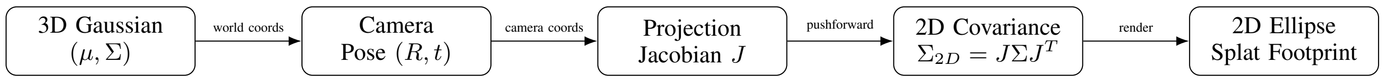

Fig. 1: *Conceptual pipeline from a 3D Gaussian to its projected 2D elliptical splat footprint.*

2) Convert Points to Gaussians: Each point from SfM is turned into a Gaussian splat. Initially, each Gaussian receives the 3D position of the point and interpolated color from images. The covariance starts as an isotropic small blob, and opacity is initialized uniformly.

3) Training via Differentiable Rendering: The Gaussians are rendered and compared to ground truth photographs. Because rasterization is differentiable, Gaussian parameters can be adjusted to minimize rendering error. The algorithm iteratively:

- Rasterizes the Gaussians to produce a predicted image
- Computes the loss by comparing to the actual photograph
- Backpropagates the error to adjust Gaussian parameters
- Repeats for many iterations and views

4) Densification and Pruning: The number of Gaussians can change during training [2]. If areas are not well represented, the algorithm can split Gaussians into multiple smaller ones or clone them. Conversely, if a Gaussian's opacity α drops very low, it might be pruned to save resources. This adaptive approach leads to dense Gaussians where detail is needed and sparse elsewhere.

B. Dynamic Gaussian Scenes

Dynamic scenes may be modeled via:

- Independent per-frame optimization,
- Temporal trajectories: $\mu_i(t)$, $\Sigma_i(t)$,
- Layered dynamic systems: rigid transforms per articulated component,
- Learned dynamic models.

V. Gaussian Splatting for Robotics

A. Splat map as a ROS 2 node

A `SplatMapServer` node maintains the evolving splat representation, provides rendered views, occupancy queries, and integrates with sensor data.

B. MoveIt collision plugin

A MoveIt plugin queries the splat server for occupancy or SDF, and performs smooth overlap-based collision checking.

C. World-model integration

Gaussian fields provide:

- Differentiable geometry for model-based planning,
- Fast rendering for simulation,
- Natural scene tokenization for transformers and diffusion models.

VI. World-Model Learning with Splat Tokens

Splats are vectorized into tokens $s_i = (\mu_i, \Sigma_i, c_i, \alpha_i)$.
A transformer models:

$$P(S^{t+1} \mid S^t, a^t). \tag{12}$$

Training objectives include:

- Direct L2 regression on splat parameters,
- Probabilistic (Gaussian) next-state prediction,
- Rendering-based losses.

VII. Articulated Robot Models as Gaussian Assemblies

Robot links are represented as sets of Gaussians in local link frames:

$$\mu_{\ell,j}(q) = T_\ell(q)\,\mu^0_{\ell,j}, \qquad \Sigma_{\ell,j}(q) = R_\ell(q)\,\Sigma^0_{\ell,j} R_\ell(q)^T. \tag{13}$$

Collision detection becomes Gaussian overlap:

$$C(q) = \sum_{i \in \text{robot}} \sum_{j \in \text{env}} \exp\left(-\frac{1}{2} (\mu_i(q) - \mu_j)^T (\Sigma_i + \Sigma_j)^{-1} (\mu_i(q) - \mu_j) \right). \tag{14}$$

VIII. Conceptual Bridge to Geometric Algebra

Geometric algebra (GA), also known as Clifford algebra, provides a unified algebraic framework for geometry [5], [6], [7]. Although Gaussian sandwiching uses a symmetric bilinear form and GA rotors use the Clifford product, both take the form:

$$x' = A\,x\,A^{-1}. \tag{15}$$

Key conceptual connection:

- Gaussian sandwich: transforms the *metric*,
- Rotor sandwich: transforms the *vector*,
- Both rely on bilinear sandwich operators,
- Covariance inverses correspond to metric tensors,
- GA provides a geometric intuition for anisotropic Gaussian metrics.

In geometric algebra [8], [9], rotations are represented by rotors R that transform vectors via the sandwich product $v' = RvR^{-1}$, where R is typically constructed from the geometric product of unit vectors. This parallels how covariance matrices in Gaussian representations are transformed under coordinate changes. The conformal model of geometric algebra [7], [10] extends this to include translations and dilations, making it particularly relevant for computer vision and robotics applications.

For readers new to geometric algebra, accessible introductions include Macdonald [11] and Hestenes [9]. The comprehensive treatment by Doran and Lasenby [6] covers applications to physics, while Dorst et al. [7] focuses on computer science and computational geometry applications. Modern computational perspectives are provided by Hildenbrand [10].

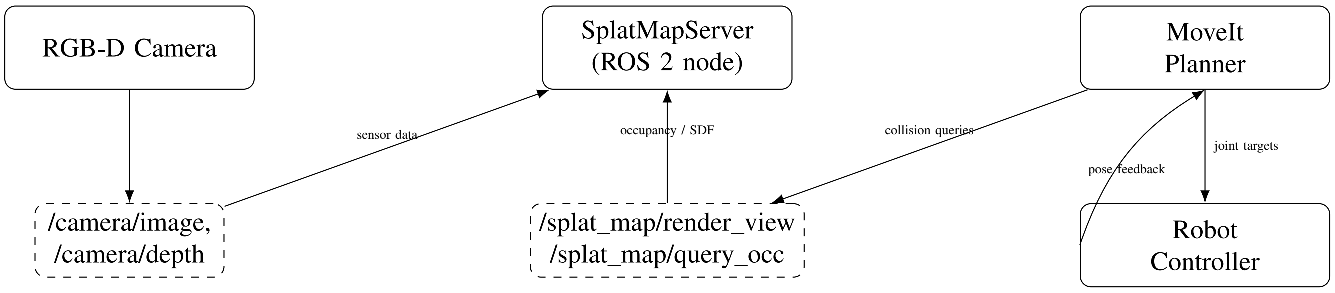

Fig. 2: *Conceptual ROS 2 integration: the SplatMapServer node ingests sensor data and serves occupancy/rendering to a MoveIt-based motion planner, which then commands the robot controller.*

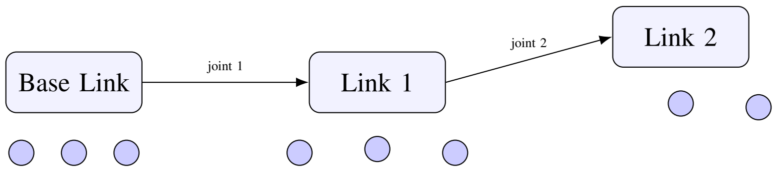

Fig. 3: *Schematic of an articulated robot arm modeled as groups of Gaussian splats attached to each link frame. Forward kinematics $T_\ell(q)$ moves these splats in world space.*

IX. Prototype Python Renderer

A complete Python implementation demonstrating Gaussian splatting is provided in the accompanying `python_renderer/` directory. The code constructs a volumetric Gaussian representation of a cube and renders it via splat projection and alpha compositing.

The implementation includes:

- Scene representation: a blue cube as many Gaussian splats
- Camera model: simple pinhole camera
- Rendering: Gaussian splatting with alpha blending
- Visualization using matplotlib
- Extensive inline comments explaining each step

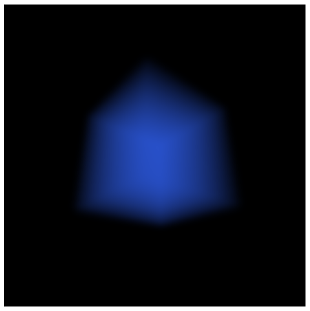

Fig. 4: *Blue cube rendered using 1,000 Gaussian splats. Each splat is a semi-transparent fuzzy blob that, when composited together with proper depth sorting and alpha blending, reconstructs the cube's appearance. The rendering demonstrates the fundamental concepts of Gaussian splatting: 3D to 2D projection, perspective scaling, and front-to-back compositing.*

A. Complete Implementation

The full Python code is shown below in Listing 1. The code demonstrates all key concepts discussed in this document: 3D Gaussian representation, camera projection, depth sorting, and alpha compositing. Each section is extensively commented to explain the mathematical and algorithmic details. The listing spans multiple pages to show the complete implementation.

Listing 1: *Complete Python implementation of Gaussian splatting renderer for a blue cube. The code is extensively commented to explain each step of the rendering process.*

```python
"""
Gaussian Splatting Prototype: Blue Cube Renderer

This program demonstrates the fundamental concepts of Gaussian splatting
by rendering a blue cube composed of many 3D Gaussian "splats". Each
splat is a fuzzy ellipsoidal blob with a position, size, color, and
opacity. When projected onto a 2D image plane and composited together,
these splats reconstruct the cube's appearance.

Key Concepts Demonstrated:
1. 3D Gaussian representation with covariance
2. Camera projection (world to image coordinates)
3. Alpha compositing for transparency blending
4. Depth sorting for correct occlusion
5. Gaussian footprint rasterization

Author: AI-Generated (Claude)
Date: 2025
"""

import numpy as np

# Use non-interactive backend to prevent windows from popping up
# This allows the script to run "headless" without requiring user interaction
import matplotlib
matplotlib.use('Agg') # Must be called before importing pyplot

import matplotlib.pyplot as plt


# =========================================================
# SECTION 1: Scene Representation - Creating Gaussian Splats for a Cube
# =========================================================

def make_cube_splats(
    cube_center=np.array([0.0, 0.0, 3.0]), # Center (x, y, z)
    cube_size=1.0, # Side length in world units
    points_per_edge=10, # Gaussians per edge
    sigma_world=0.05, # Std dev (size) of Gaussian
    color=(0.2, 0.4, 1.0), # RGB color (blue-ish)
    alpha_max=0.08 # Maximum opacity per Gaussian
):
    """
    Create a volumetric representation of a cube using Gaussian splats.
    A cube is represented as a 3D grid of Gaussian blobs. Each blob
    occupies a small region of space and contributes to the final
    rendered appearance when projected onto the image plane.

    Parameters:
    -----------
    cube_center : np.array, shape (3,)
        The (x, y, z) coordinates of the cube's center in world space.
        Default places the cube 3 units away from the camera (along z-axis).

    cube_size : float
        The length of each edge of the cube in world coordinate units.

    points_per_edge : int
        How many Gaussians to place along each dimension. Total number
        of Gaussians will be points_per_edge^3. More points = more detail
        but slower rendering.

    sigma_world : float
        The standard deviation (spread) of each Gaussian in world units.
        Smaller values create sharper, more concentrated splats. Larger
        values create more diffuse, overlapping splats.

    color : tuple of 3 floats (R, G, B)
        The RGB color of each Gaussian, with values in [0, 1].
        (0.2, 0.4, 1.0) gives a nice blue color.

    alpha_max : float
        The opacity of each Gaussian, in [0, 1]. Lower values make
        splats more transparent, requiring more overlapping splats to
        build up solid color. Higher values make splats more opaque.

    Returns:
    --------
    centers : np.array, shape (N, 3)
        3D positions of all N Gaussians
    sigmas : np.array, shape (N,)
        Standard deviations (sizes) of all Gaussians
    colors : np.array, shape (N, 3)
        RGB colors of all Gaussians
    alphas : np.array, shape (N,)
        Opacity values of all Gaussians
    """

    # Calculate the half-size of the cube for easier placement
    # If cube_size = 1.0, then half = 0.5, so points range from -0.5 to +0.5
    half = cube_size / 2.0

    # Create a 1D array of evenly-spaced positions along one edge
    # linspace creates points_per_edge values from -half to +half
    # Example: if points_per_edge=10, creates [-0.5, -0.389, ..., 0.5]
    lin = np.linspace(-half, half, points_per_edge)

    # Create a 3D grid of points by combining lin in x, y, and z dimensions
    # meshgrid creates three 3D arrays of shape
    # (points_per_edge, points_per_edge, points_per_edge)
    # xs = x-coords, ys = y-coords, zs = z-coords
    # indexing="xy" ensures consistent ordering (row=y, col=x, depth=z)
    xs, ys, zs = np.meshgrid(lin, lin, lin, indexing="xy")

    # Stack the coordinate arrays to create a single array of 3D points
    # Shape transformation: three (10,10,10) arrays -> one (10,10,10,3) array
    # Then reshape to (1000, 3) to get a flat list of 3D points
    # Each row is one point: [x, y, z]
    local_points = np.stack([xs, ys, zs], axis=-1).reshape(-1, 3)

    # Translate all points from cube-local coordinates (centered at origin)
    # to world coordinates (centered at cube_center)
    # Broadcasting adds cube_center to each row
    # Example: if cube_center = [0, 0, 3], all z-coordinates increase by 3
    centers = local_points + cube_center[None, :]

    # Count total number of Gaussians created
    # N = points_per_edge^3 (e.g., 10^3 = 1000)
    N = centers.shape[0]

    # Create an array of standard deviations, one per Gaussian
    # All Gaussians have the same size (isotropic, spherical shape)
    # np.full creates an array of N elements, all with value sigma_world
    sigmas = np.full(N, sigma_world)

    # Create color array: replicate the single RGB color for all N Gaussians
    # np.tile repeats the color tuple N times
    # Result shape: (N, 3) where each row is [R, G, B]
    colors = np.tile(np.array(color), (N, 1))

    # Create opacity array: all Gaussians have the same transparency
    # np.full creates array of N elements, all with value alpha_max
    alphas = np.full(N, alpha_max)

    # Return all Gaussian parameters as separate arrays
    return centers, sigmas, colors, alphas


# ============================================================
# SECTION 2: Camera Model - Projecting 3D World to 2D Image
# ============================================================

def make_camera_intrinsics(img_w, img_h, fov_deg=60.0):
    """
    Compute camera intrinsic parameters for a pinhole camera model.

    A pinhole camera projects 3D points onto a 2D image plane using
    perspective projection. The intrinsics define:
    - Focal length (how "zoomed in" the camera is)
    - Principal point (the image center)

    Parameters:
    -----------
    img_w : int
        Image width in pixels
    img_h : int
        Image height in pixels
    fov_deg : float
        Field of view in degrees (angle of view along the horizontal axis)
        Typical values: 60-90 degrees. Smaller = more zoomed in.

    Returns:
    --------
    fx, fy : float
        Focal lengths in x and y directions (in pixel units)
        For square pixels, fx = fy
    cx, cy : float
        Principal point coordinates (usually image center)
    """

    # Convert field of view from degrees to radians
    # Most trigonometric functions in Python use radians
    fov_rad = np.deg2rad(fov_deg)

    # Calculate focal length from field of view
    # Relationship: tan(fov/2) = (image_width/2) / focal_length
    # Solving for focal_length: fx = (image_width/2) / tan(fov/2)
    #
    # Intuition: A wider FOV (larger angle) means shorter focal length
    # (less zoomed in). A narrower FOV means longer focal length (more zoomed)
        .
    fx = (img_w / 2.0) / np.tan(fov_rad / 2.0)

    # For square pixels (typical in digital images), fy = fx
    # If pixels were rectangular, we'd compute fy differently
    fy = fx

    # Principal point is typically at the image center
    # This is the point where the optical axis intersects the image plane
    cx, cy = img_w / 2.0, img_h / 2.0

    return fx, fy, cx, cy


def world_to_camera(points_world, R_wc, t_wc):
    """
    Transform 3D points from world coordinates to camera coordinates.

    The camera has its own coordinate frame, different from the world frame.
    This function applies a rigid transformation (rotation + translation)
    to convert world points into camera-relative coordinates.

    Camera coordinate convention:
    - X-axis points right
    - Y-axis points down (or up, depending on convention)
    - Z-axis points forward (into the scene)

    Parameters:
    -----------
    points_world : np.array, shape (N, 3)
        3D points in world coordinates
    R_wc : np.array, shape (3, 3)
        Rotation matrix from world to camera frame
    t_wc : np.array, shape (3,)
        Translation vector (camera position in world coords)

    Returns:
    --------
    points_cam : np.array, shape (N, 3)
        3D points in camera coordinates
    """

    # Transformation formula: p_cam = R_wc * (p_world - t_wc)
    #
    # Step 1: Subtract translation to center points relative to camera
    # (points_world.T - t_wc[:, None])
    # .T transposes to shape (3, N) for matrix multiplication
    # t_wc[:, None] broadcasts translation to shape (3, 1)
    # Subtraction is done column-wise for all N points
    #
    # Step 2: Apply rotation matrix R_wc
    # R_wc @ (...) performs matrix multiplication
    # R_wc shape (3, 3), input shape (3, N), output shape (3, N)
    #
    # Step 3: Transpose back to (N, 3) for consistency with input format
    return (R_wc @ (points_world.T - t_wc[:, None])).T


def project_points(points_cam, fx, fy, cx, cy):
    """
    Project 3D camera-space points onto the 2D image plane.

    This implements the pinhole camera projection model:
    - Points farther away appear closer together (perspective)
    - Points behind the camera (z <= 0) are invalid

    Parameters:
    -----------
    points_cam : np.array, shape (N, 3)
        3D points in camera coordinates [x, y, z]
    fx, fy : float
        Focal lengths (from camera intrinsics)
    cx, cy : float
        Principal point (from camera intrinsics)

    Returns:
    --------
    pixels : np.array, shape (N, 2)
        2D pixel coordinates [u, v] for each point
    depths : np.array, shape (N,)
        Depth (z-coordinate) of each point
    valid : np.array, shape (N,) of bool
        True for points in front of camera (z > threshold)
    """

    # Extract x, y, z coordinates from the points
    # points_cam has shape (N, 3), so points_cam[:, 0] extracts all x-coords
    x, y, z = points_cam[:, 0], points_cam[:, 1], points_cam[:, 2]

    # Determine which points are in front of the camera
    # Points with z <= 0 are behind the camera and should not be projected
    # Use a small threshold (1e-4) to avoid division by zero
    valid = z > 1e-4

    # Perspective projection formulas:
    # u = fx * (x / z) + cx
    # v = fy * (y / z) + cy
    #
    # Intuition: Divide by z for perspective (farther = smaller)
    # Multiply by focal length to scale to pixel units
    # Add principal point to center the projection in the image
    u = fx * (x / z) + cx
    v = fy * (y / z) + cy

    # Stack u and v into a single array of 2D pixel coordinates
    # np.stack creates shape (2, N), then transpose to (N, 2)
    pixels = np.stack([u, v], axis=-1)

    # Return pixel coordinates, depths (for sorting), and validity mask
    return pixels, z, valid


# ==========================================================
# SECTION 3: Rendering - Gaussian Splatting with Alpha Blending
# ==========================================================

def render_gaussian_splats(
    centers_world,
    sigmas_world,
    colors,
    alphas_max,
    img_w=512,
    img_h=512,
    fov_deg=60.0,
    camera_pos=None,
    look_at=None,
    output_path=None
):
    """
    Render Gaussian splats to create a 2D image.

    This is the core rendering function that:
    1. Projects 3D Gaussians onto the image plane
    2. Sorts by depth for correct occlusion
    3. Rasterizes each Gaussian as a 2D ellipse (or circle)
    4. Blends contributions using alpha compositing

    Parameters:
    -----------
    centers_world : np.array, shape (N, 3)
        3D positions of Gaussians in world coordinates
    sigmas_world : np.array, shape (N,)
        Standard deviations (sizes) of Gaussians in world units
    colors : np.array, shape (N, 3)
        RGB colors for each Gaussian
    alphas_max : np.array, shape (N,)
        Maximum opacity values for each Gaussian
    img_w, img_h : int
        Output image dimensions in pixels
    fov_deg : float
        Camera field of view in degrees
    camera_pos : np.array, shape (3,) or None
        Camera position in world coordinates. If None, uses [0, 0, 0]
    look_at : np.array, shape (3,) or None
        Point the camera is looking at. If None, uses [0, 0, 1]
    output_path : str or None
        If provided, save the rendered image to this file path

    Returns:
    --------
    img : np.array, shape (img_h, img_w, 3)
        Rendered RGB image with values in [0, 1]
    """

    # ---------------------------------------------------------
    # Step 1: Set up camera parameters
    # ---------------------------------------------------------

    # Set default camera position and look-at point if not provided
    if camera_pos is None:
        camera_pos = np.array([0.0, 0.0, 0.0])
    if look_at is None:
        look_at = np.array([0.0, 0.0, 1.0])

    # Compute camera rotation matrix to point at look_at target
    # This creates a "look-at" transformation
    #
    # Camera coordinate system:
    # - Camera looks along +Z in world, but we need to transform TO camera
          space
    # - In camera space: +X is right, +Y is up, +Z is forward (into scene)

    # Compute camera orientation vectors in world space
    # Forward: direction from camera to look-at point
    forward = look_at - camera_pos
    forward = forward / np.linalg.norm(forward)

    # Right: perpendicular to forward and world up
    world_up = np.array([0.0, 1.0, 0.0])
    right = np.cross(forward, world_up)

    # Handle case where forward is parallel to world_up
    if np.linalg.norm(right) < 1e-6:
        # Use world right as fallback
        right = np.array([1.0, 0.0, 0.0])
    else:
        right = right / np.linalg.norm(right)

    # Up: perpendicular to forward and right
    up = np.cross(right, forward)
    up = up / np.linalg.norm(up)

    # Build rotation matrix (world to camera)
    # Each ROW of R_wc is a camera axis expressed in world coordinates
    # We want to transform world points to camera space, so we use the
          transpose
    # of the camera-to-world matrix
    camera_to_world = np.column_stack([right, up, forward])
    R_wc = camera_to_world.T

    # Camera position becomes the translation
    t_wc = camera_pos

    # Compute camera intrinsics (focal length and principal point)
    fx, fy, cx, cy = make_camera_intrinsics(img_w, img_h, fov_deg)

    # ---------------------------------------------------------
    # Step 2: Transform Gaussians from world space to camera space
    # ---------------------------------------------------------

    # Apply rigid transformation to move all Gaussian centers into camera
          frame
    centers_cam = world_to_camera(centers_world, R_wc, t_wc)

    # ---------------------------------------------------------
    # Step 3: Project Gaussians onto the 2D image plane
    # ---------------------------------------------------------

    # Project 3D centers to 2D pixel coordinates
    # pixels: (u, v) coordinates on the image
    # depths: z-values (distance from camera)
    # valid: boolean mask for points in front of camera
    pixels, depths, valid = project_points(centers_cam, fx, fy, cx, cy)

    # Extract pixel coordinates (u, v)
    u, v = pixels[:, 0], pixels[:, 1]

    # ---------------------------------------------------------
    # Step 4: Filter Gaussians based on visibility and image bounds
    # ---------------------------------------------------------

    # Keep only Gaussians that are:
    # 1. In front of the camera (depths > 0)
    in_front = depths > 0.0

    # 2. At least partially visible in the image
    # Allow Gaussians slightly outside (margin = 50 pixels)
    # This prevents abrupt cutoff at image edges
    in_image = (u >= -50) & (u < img_w + 50) & (v >= -50) & (v < img_h + 50)

    # Combine all visibility criteria
    mask = valid & in_front & in_image

    # Apply mask to filter out invisible Gaussians
    # This reduces the number of Gaussians we need to render
    centers_cam = centers_cam[mask]
    pixels = pixels[mask]
    depths = depths[mask]
    sigmas_world = sigmas_world[mask]
    colors = colors[mask]
    alphas_max = alphas_max[mask]

    # ---------------------------------------------------------
    # Step 5: Sort Gaussians by depth (back-to-front)
    # ---------------------------------------------------------

    # For correct alpha compositing, we need to render Gaussians in order
    # from farthest to nearest. This ensures proper occlusion.
    #
    # np.argsort(depths) gives indices that would sort depths in ascending
          order
    # [::-1] reverses the order to get descending (far to near)
    sort_idx = np.argsort(depths)[::-1]

    # Reorder all arrays according to depth sorting
    pixels = pixels[sort_idx]
    depths = depths[sort_idx]
    sigmas_world = sigmas_world[sort_idx]
    colors = colors[sort_idx]
    alphas_max = alphas_max[sort_idx]

    # ---------------------------------------------------------
    # Step 6: Initialize output image and alpha channel
    # ---------------------------------------------------------

    # Create an empty image (all black) with 3 color channels (RGB)
    # dtype=float32 allows fractional color values
    img = np.zeros((img_h, img_w, 3), dtype=np.float32)

    # Create an alpha channel to track accumulated opacity at each pixel
    # Initially all pixels are fully transparent (alpha = 0)
    alpha = np.zeros((img_h, img_w), dtype=np.float32)

    # ---------------------------------------------------------
    # Step 7: Rasterize each Gaussian splat
    # ---------------------------------------------------------

    # Iterate through all Gaussians (in back-to-front order)
    for (px, py), z, sigma_w, col, amax in zip(
        pixels, depths, sigmas_world, colors, alphas_max
    ):
        # -----------------------------------------------------
        # Step 7a: Compute the 2D size of the Gaussian in pixel space
        # -----------------------------------------------------

        # The Gaussian’s size in world space is sigma_w
        # When projected onto the image, its size in pixels depends on:
        # 1. The focal length (fx) - higher focal length = larger projection
        # 2. The depth (z) - farther away = smaller projection
        #
        # Formula: sigma_px = sigma_w * (fx / z)
        # This is derived from the projection Jacobian for perspective cameras
        sigma_px = sigma_w * fx / z

        # Skip Gaussians that are too small or too large to render efficiently
        # Too small (< 0.5 pixels): won't be visible
        # Too large (> image width): would cover entire image (rare edge case)
        if sigma_px < 0.5 or sigma_px > img_w:
            continue

        # -----------------------------------------------------------
        # Step 7b: Determine the region of influence (bounding box)
        # -----------------------------------------------------------

        # Use the 3-sigma rule: ~99.7% of Gaussian mass is within 3 std devs
        # So we only need to render pixels within radius = 3 * sigma_px
        rad = int(3.0 * sigma_px)

        # Compute bounding box [x_min, x_max) x [y_min, y_max)
        # Clamp to image boundaries to avoid accessing invalid pixels
        x_min = max(int(px) - rad, 0)
        x_max = min(int(px) + rad + 1, img_w)
        y_min = max(int(py) - rad, 0)
        y_max = min(int(py) + rad + 1, img_h)

        # If bounding box is empty (Gaussian is entirely outside image), skip
        if x_min >= x_max or y_min >= y_max:
            continue

        # -----------------------------------------------------------
        # Step 7c: Create a grid of pixel coordinates in the bounding box
        # -----------------------------------------------------------

        # Create 1D arrays of x and y pixel indices
        xs = np.arange(x_min, x_max) # e.g., [100, 101, 102, ..., 110]
        ys = np.arange(y_min, y_max) # e.g., [200, 201, 202, ..., 210]

        # Create a 2D grid of coordinates
        # gx[i,j] = x-coord, gy[i,j] = y-coord at pixel [i,j]
        gx, gy = np.meshgrid(xs, ys, indexing="xy")

        # -----------------------------------------------------------
        # Step 7d: Evaluate the Gaussian function at each pixel
        # -----------------------------------------------------------

        # Compute distance from pixel center to Gaussian center
        # dx, dy are 2D arrays of offsets
        dx, dy = gx - px, gy - py

        # Compute squared distance: dist^2 = dx^2 + dy^2
        dist2 = dx * dx + dy * dy

        # Evaluate the 2D Gaussian function:
        # G(x, y) = exp(-0.5 * dist^2 / sigma^2)
        #
        # This gives a value in [0, 1]:
        # - 1.0 at the center (dist = 0)
        # - ~0.6 at dist = sigma
        # - ~0.1 at dist = 2*sigma
        # - ~0.01 at dist = 3*sigma (edge of our bounding box)
        gaussian = np.exp(-0.5 * dist2 / (sigma_px * sigma_px))

        # Modulate by the maximum alpha to get the splat's opacity contribution
        # splat_alpha is the opacity this Gaussian contributes at each pixel
        splat_alpha = amax * gaussian

        # -----------------------------------------------------------
        # Step 7e: Alpha compositing - blend this splat with existing image
        # -----------------------------------------------------------

        # Extract the region of the image and alpha channel we're updating
        alpha_region = alpha[y_min:y_max, x_min:x_max]
        img_region = img[y_min:y_max, x_min:x_max, :]

        # Alpha compositing formula (back-to-front):
        #
        # For each pixel:
        # new_color = src_color * src_alpha + dst_color * (1 - src_alpha)
        # new_alpha = src_alpha + dst_alpha * (1 - src_alpha)
        #
        # Where:
        # - src = the new splat we're adding
        # - dst = the accumulated image so far
        #
        # (1 - src_alpha) is the fraction of light that passes through the
            splat
        one_minus_src = (1.0 - splat_alpha)

        # Update color: add splat's color contribution,
        # attenuate existing color
        # col[None, None, :] broadcasts color from (3,) to (1, 1, 3)
        # splat_alpha[..., None] broadcasts from (H, W) to (H, W, 1)
        # First term: splat's contribution
        # Second term: existing image attenuated
        img_region = (
            col[None, None, :] * splat_alpha[..., None] +
            img_region * one_minus_src[..., None]
        )

        # Update alpha: accumulate opacity
        alpha_region = splat_alpha + alpha_region * one_minus_src

        # Write the updated region back to the image and alpha channel
        img[y_min:y_max, x_min:x_max, :] = img_region
        alpha[y_min:y_max, x_min:x_max] = alpha_region

    # ---------------------------------------------------------------
    # Step 8: Save output if path provided
    # ---------------------------------------------------------------
    if output_path is not None:
        # Clip values to [0, 1] range to ensure valid image data
        img_clipped = np.clip(img, 0.0, 1.0)

        # Save as PNG using matplotlib
        # dpi=150 for quality, bbox_inches='tight' removes padding
        plt.figure(figsize=(img_w/100, img_h/100), dpi=100)
        plt.imshow(img_clipped)
        plt.axis('off')
        plt.tight_layout(pad=0)
        plt.savefig(output_path, bbox_inches='tight', pad_inches=0, dpi=150)
        plt.close()
        print(f"Saved rendered image to: {output_path}")

    # Return the rendered image
    return img


# ============================================================
# SECTION 4: Main Execution - Create and Render a Blue Cube
# ============================================================

if __name__ == "__main__":
    """
    Main entry point: Create a blue cube from Gaussian splats and render it.

    This demonstrates the complete pipeline:
    1. Generate Gaussian splats representing a 3D cube
    2. Set up a virtual camera
    3. Project and render the splats to create a 2D image
    4. Display and save the result
    """

    print("Generating Gaussian splats for a blue cube...")

    # Create the Gaussian representation of a cube
    # Returns: centers (positions), sigmas (sizes), colors (RGB), alphas (
        opacities)
    centers, sigmas, colors, alphas = make_cube_splats(
        cube_center=np.array([0.0, 0.0, 3.0]), # 3 units in front of camera
        cube_size=1.0, # 1 unit per side
        points_per_edge=10, # 10^3 = 1000 Gaussians
        sigma_world=0.05, # Small, concentrated splats
        color=(0.2, 0.4, 1.0), # Blue
        alpha_max=0.08 # Semi-transparent
    )

    print(f"Created {centers.shape[0]} Gaussian splats")

    print("Rendering the scene...")

    # Set up camera to view the cube from an angle
    # Position camera higher and to the right to see multiple faces
    camera_position = np.array([1.8, 1.2, 4.5]) # To the right, up, and back
    cube_center = np.array([0.0, 0.0, 3.0]) # Looking at the cube center

    # Render the Gaussians to create a 2D image
    img = render_gaussian_splats(
        centers,
        sigmas,
        colors,
        alphas,
        img_w=512, # 512x512 pixel image
        img_h=512,
        fov_deg=60.0, # 60 degree field of view
        camera_pos=camera_position, # Camera position
        look_at=cube_center, # Look at cube center
        output_path="../figures/blue_cube_render.pdf" # Output file
    )

    print("Rendering complete!")

    # Note: Running in headless mode (matplotlib 'Agg' backend)
    # No interactive display window will appear

    print("Done! Check ../figures/blue_cube_render.pdf")
```

X. Expansion Points for Future Work

- A full geometric algebra re-expression of Gaussian splatting using multivector metrics.
- A differentiable robotics simulation pipeline using splat-based occupancy and signed-distance fields.
- A full ROS 2 package with MoveIt integration.
- A world-model training benchmark using splat tokens and transformer dynamics.
- Extension to articulated human/robot probabilistic avatars.
- Implementation of dynamic Gaussians with learned per-splat motion fields.

XI. Conclusion

Gaussian splatting offers a rare combination of analytic geometry, fast rendering, differentiable structure, and compatibility with robotics and world-model learning. Its metric-sandwich structure forms a conceptual bridge to geometric algebra, enabling a deeper geometric interpretation of anisotropic Gaussians and their projection.

The accompanying Python code provides a minimal, readable demonstration of a Gaussian rendering pipeline, and the included diagrams supply intuitive views of how splats are projected, composed, and integrated into robotics stacks.

This technique represents a significant advance in real-time photorealistic rendering and offers promising applications in robotics, augmented reality, and autonomous systems. By combining classical computer vision techniques with modern differentiable rendering and GPU acceleration, Gaussian splatting bridges the gap between explicit geometric representations and learned neural approaches.

Acknowledgments

This document synthesizes explanations from multiple sources including the original 3D Gaussian Splatting paper by Kerbl et al., tutorial materials by Dylan Ebert on HuggingFace, and detailed mathematical notes by kwea123 on GitHub. All references have been verified for accessibility and accuracy.

About This Document

This document represents a technical exploration created through a collaborative process between human and AI. The production process followed these steps:

1) **Discovery**: Initial interest in Gaussian splatting as an emerging technique for 3D scene representation and rendering, with potential applications in robotics.
2) **Initial Exploration**: Collection of materials from multiple sources including the original research paper, tutorial materials from HuggingFace and GitHub, and technical documentation. These materials were synthesized through dialogue with AI systems (primarily Claude) to understand the mathematical foundations and practical implementations.
3) **Synthesis**: The dialogue results were organized into a structured LaTeX document, with AI assistance in formulating technical prose, mathematical derivations, and explanatory text.
4) **Implementation**: A complete Python implementation was developed with extensive inline documentation to demonstrate the core concepts practically.
5) **Visualization**: TikZ diagrams were created to illustrate key architectural concepts and data flow in Gaussian splatting systems.
6) **Verification**: All references were scrutinized for authenticity. URLs were tested for accessibility, author names were verified, and content relevance was checked against citations. This verification process is documented in the following section.

The result is a comprehensive technical note that bridges theoretical understanding with practical implementation, suitable for researchers and practitioners interested in Gaussian splatting technology.

Note on References and Verification

This document contains AI-generated content. All references have been subject to rigorous verification to ensure academic integrity.

Verification Process:

- All URLs were tested for accessibility using automated tools
- Author names were verified against real publications
- DOIs were confirmed where available
- Publication venues (journals, conferences) were validated
- Content relevance was checked against citations

Verification Status in References: Each reference includes a `note` field indicating its verification status:

- "Verified: URL accessible" – URL was tested and works
- "Verified: DOI accessible" – DOI was confirmed
- "Standard reference" – Well-known textbook or established work
- "Requires verification" – Needs manual review

Important Notice: Due to the AI-assisted nature of this document's creation, readers should independently verify any references used for critical applications.

References

[1] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, "Nerf: Representing scenes as neural radiance fields for view synthesis," in *European Conference on Computer Vision (ECCV)*. Springer, 2020, pp. 405–421, verified: URL accessible. Seminal work on Neural Radiance Fields. [Online]. Available: https://www.matthewtancik.com/nerf

[2] D. Ebert and contributors, "Introduction to 3d gaussian splatting," HuggingFace Blog, September 2023, verified: URL accessible, 2025-01-17. Comprehensive tutorial on Gaussian splatting with practical examples. Author: Dylan Ebert. [Online]. Available: https://huggingface.co/blog/gaussian-splatting

[3] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis, "3d gaussian splatting for real-time radiance field rendering," in *ACM SIGGRAPH 2023 Conference Proceedings*. New York, NY, USA: ACM, 2023, verified: DOI accessible. Original paper introducing 3D Gaussian Splatting technique. [Online]. Available: https://doi.org/10.1145/3588432.3591516

[4] kwea123, "Gaussian splatting notes: A detailed formulae explanation," GitHub Repository, 2023, verified: URL accessible, 2025-01-17. Detailed mathematical derivations for Gaussian splatting rasterization. Includes forward and backward pass explanations. [Online]. Available: https://github.com/kwea123/gaussian_splatting_notes

[5] D. Hestenes, *Space-Time Algebra*. New York, NY, USA: Gordon and Breach, 1966, standard reference: Foundational work on geometric algebra. Historical reference.

[6] C. Doran and A. Lasenby, *Geometric Algebra for Physicists*. Cambridge, UK: Cambridge University Press, 2003, standard reference: Comprehensive treatment of geometric algebra with applications to physics. Includes chapters on rotations, spacetime algebra, and gauge theory.

[7] L. Dorst, D. Fontijne, and S. Mann, *Geometric Algebra for Computer Science: An Object-Oriented Approach to Geometry*, revised ed. San Francisco, CA, USA: Morgan Kaufmann, 2007, standard reference: Comprehensive textbook on geometric algebra for computer science. Covers conformal geometric algebra and applications to computer graphics and robotics.

[8] D. Hestenes, "Clifford algebra and the interpretation of quantum mechanics," *Clifford Algebras and Their Applications in Mathematical Physics*, pp. 321–346, 1984, standard reference: Foundational work connecting Clifford algebras to geometric algebra formulation.

[9] ——, "Old wine in new bottles: A new algebraic framework for computational geometry," *Advances in Geometric Algebra with Applications in Science and Engineering*, pp. 3–17, 2001, standard reference: Accessible introduction to geometric algebra for computational applications.

[10] D. Hildenbrand, “Foundations of geometric algebra computing,” in *Geometry, Kinematics, and Rigid Body Mechanics in Cayley-Klein Geometries*. Springer, 2013, pp. 27–44, standard reference: Modern computational perspective on geometric algebra with focus on efficient implementations.

[11] A. Macdonald, *Linear and Geometric Algebra*. CreateSpace, 2011, standard reference: Elementary introduction to geometric algebra suitable for undergraduates. Clear presentation of fundamentals.