Course Report for Visual Media: 3D Gaussian Splatting for Real-Time Radiance Field Rendering [1]

Junqing Ma

July 31, 2024

1 Introduction

Since the emergence of 3D Gaussian Splatting (3DGS) [1], the entire NeRF community has been stirred up. The most attractive aspect is that it can be integrated with traditional rasterization while maintaining high reconstruction quality and fast optimization speed, and abandon the heavy neural network for light-wight training and editable representation. Overall, it can be seen as a culmination of various innovations.

For recent weeks, I have studied the fundamental principles of 3D Gaussians and compares them with traditional explicit and implicit geometries. Furthermore, I have reimplemented the representation and rasterization part of it, and I am willing to share my understanding on it.

2 How to express implicit geometry

Let's review the classic NeRF-based methods. NeRF [2] is inspired by the observation that in the reality, we see an object means that the light originating from the surface hits human eyes. From a physics perspective, light is a type of electromagnetic radiation field that permeates all of space. This electromagnetic radiation contains all the information related to imaging, such as color and intensity. Therefore, we can use a convolutional neural network to implicitly represent the field equations of this entire space. By sampling at as many spatial points and angles as possible, we can train a representation that is as accurate as possible. This is the core idea behind neural radiance fields (NeRF).

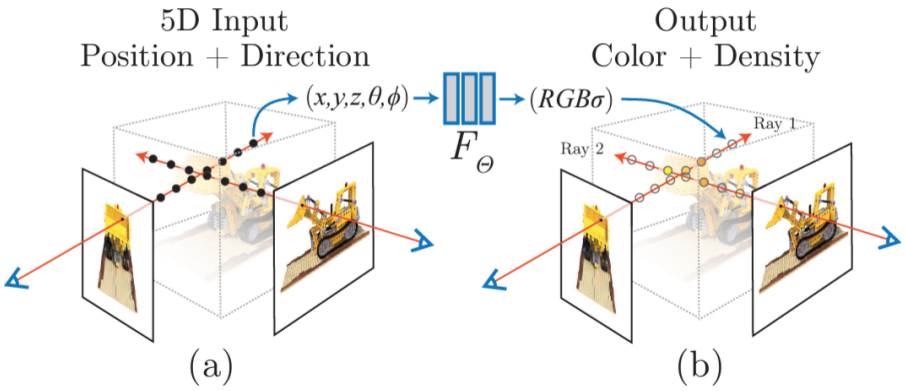

Figure 1: NeRF pipeline

The most classic NeRF (density and color) belongs to volumetric representation, where the density value returned by the sampling points indicates the presence of geometric occupancy. Another popular approach is surface representation. For surface representation, given an input sampling point, the SDF (Signed Distance Function) outputs the distance from this point to the nearest surface in the space. A positive value indicates the outside of the surface, while a negative value indicates the inside. The surface can be represented by the set of all sampling points where SDF equals zero, and a neural network can be used to realize SDF, known as neural SDF.

Volumetric methods determine the color contribution of a sampling point based on geometric density, while surface methods consider that sampling points closer to the surface contribute more to the color.

3 Why Gaussians, not other representations?

Depite the achievement from NeRF, the limitation is much more dazzling. The training time is very long, which may take 1 2 days to wait for a high-quality reconstruction. Besides, the implicit representation makes the data structure a pure black box, and we have no idea how to directly manipulate and modify it. Sampling on the neural network can't reach real-time performance, either. It still takes a few seconds to synthesize image in a fixed direction, which is far from 60FPS, the standard of real-time baseline.

Compared to NeRF, 3D Gaussians (3DGS) have numerous advantages. They are based on rasterization rather than ray tracing, which significantly reduces the computational load for rendering. The implicit representation of NeRF makes it uneditable, whereas the Gaussian ellipsoids in 3DGS can be freely added, removed, translated, and rotated. This greatly enhances the editability of scenes.

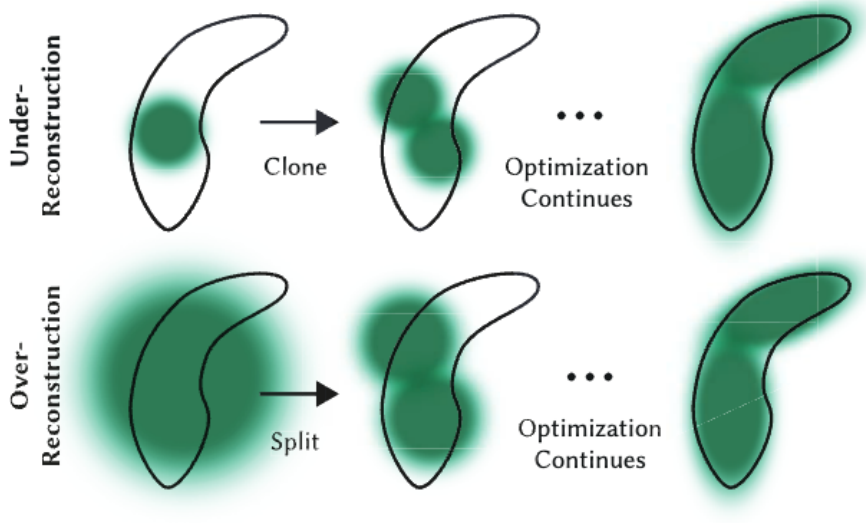

Figure 2: In the training process, split or merge Gaussians to represent shapes.

Moreover, using Gaussian ellipsoids to model objects and then rendering them with splatting transformations is not a recent idea. As early as 20 years ago, there were attempts to sample local surfaces using Gaussian kernels to simulate them, resulting in high-quality reconstruction. Voxel splatting algorithms based on Gaussian kernels have also long been established. Even in earlier times, we could see games using ellipsoids as primitives. For example, the 1994 game Ecstatica [3] used ellipsoids as its basic elements.

Figure 3: Screenshot for Ecstatica

A single 3D Gaussian can be optimized as a small differentiable space, while different Gaussians can be rasterized in parallel, similar to triangles. This represents a subtle balance between differentiability and discreteness. Next, I will introduce what I have implemented in 3D Gaussian Splatting.

4 3D Gaussian

4.1 From 1D to 3D Gaussian

We are familiar with 1D Gauss function:

$$N_{\mu,\sigma}(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{1}$$

The image shows a single bell-shaped symmetric curve. The mean μ controls the axis of symmetry, thus determining the position of the curve, while the standard deviation σ controls the concentration of density. Integrating over a certain interval can yield the probability that data within the distribution falls in that interval, with the majority of the data falling within the range $[\mu - 3\sigma, \mu + 3\sigma]$ of the probability 0.9974.

A set of parameters μ, σ can determine a 1D Gaussian distribution function, which in turn determines a 1D line segment $[\mu-3\sigma, \mu+3\sigma]$. By changing the parameters μ, σ, it can represent a line segment on the 1D number line. Similarly, when extended to three dimensions, a 3D Gaussian distribution determines an ellipsoid. This ellipsoid is symmetric , and the cross-sections cut from planes perpendicular to the axes are ellipses (or circles). However, since this ellipsoid can rotate and move, its symmetry axes (hereafter referred to as the model coordinate system) do not necessarily align with the world coordinate system.

The simplest example is as following, whose axes are parallel with xyz axes:

$$N_{\mu_x,\sigma_x,\mu_y,\sigma_y,\mu_z,\sigma_z}(x,y,z) = \frac{1}{(\sqrt{2\pi})^3\sigma_x\sigma_y\sigma_z} \exp\left(-\frac{(x-\mu_x)^2}{2{\sigma_x}^2} - \frac{(y-\mu_y)^2}{2{\sigma_y}^2} - \frac{(z-\mu_z)^2}{2{\sigma_z}^2}\right) \tag{2}$$

4.2 3D Gaussians

Just as three vertices can represent any 3D triangle, researchers naturally hope that the basic elements they construct can cover a sufficiently diverse range of geometries. Here, we first present the standard form of 3D Gaussians, which can encompass ellipsoids of any shape in space (including translation and rotation):

$$G_s(x) = \left(\frac{1}{\sqrt{2\pi}^3 \det(\Sigma)}\right) e^{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)} \tag{3}$$

You can observe some common structures in Gaussian distributions, such as the form of the normalization coefficient (which normalizes the density integral to 1) multiplied by the exponential term. Here, x is the 3D column vector coordinate, μ is the center of the ellipsoid (which controls translation in world space), and the covariance matrix Σ controls the stretching and rotation of the ellipsoid along the three axes (model space). The eigenvectors of the covariance matrix are the symmetry axes of the ellipsoid.

The paper defines it as follows:

$$G(x) = e^{-\frac{1}{2}(x)^T\Sigma^{-1}(x)} \tag{4}$$

Comparing with the standard form, you can see that the scale factor in front of the exponential part has been removed (which does not affect the ellipsoid's geometry); it is assumed that the model's coordinate center is at the origin for convenience in rotation and scaling, and translation is added when placed in world space.

4.3 Initialize the Ellipsoid

The above explanation describes how to define an ellipsoid using a 3D Gaussian, but it doesn't clarify how to create an ellipsoid from scratch. In the original paper, adaptively generating new ellipsoids is a crucial step. Simply randomly generating numbers to fill the covariance matrix Σ is insufficient, as each 3D Gaussian ellipsoid corresponds to a Σ, but not every Σ can correspond to an ellipsoid. Σ must be positive semi-definite.

Since a numerical approach is challenging, it's better to approach it geometrically. The original paper mentions that 3D Gaussians and ellipsoids are isomorphic, and an ellipsoid can be obtained by scaling a sphere along its axes and then rotating it (note the order: scale first, then rotate, as the ellipsoid's symmetry axes always correspond to the model space's xyz axes, which are mutually perpendicular). Therefore, the paper proposes the initialization method of using a scaling transformation S and a rotation transformation R to combine and obtain Σ:

$$\Sigma = RSS^T R^T \tag{5}$$

Since the scaling transformations are all along the coordinate axes, only a 3D vector s is needed. The rotation is represented using a quaternion q. In machine learning, parameters are typically optimized using gradient descent, but directly optimizing Σ may not ensure that it remains positive semi-definite. Therefore, the paper's method continues to propagate the gradient down to s and q for optimization.

As for how to adaptively generate new ellipsoids during optimization and assign attributes to them, that's a more complex story. This report lays the groundwork as an introduction.

5 Splatting

With a basic understanding of 3D Gaussians, let's move on to Splatting. One of the main aspects of traditional rasterization is mapping 3D triangles onto the projection plane and then pixelating them. In modern GPU pipelines, this process is typically not exposed, with the graphics pipeline handling vertices and assembly before rasterization and fragments and pixels afterward, leaving the intermediate steps to the GPU.

The ellipsoids discussed earlier are all 3D geometries. For the rasterization of ellipsoids, developers must implement this using GPGPU. When an ellipsoid is projected onto the projection plane, the resulting 2D shape is called a "Splat," similar to the mark left by a snowball hitting a wall.

5.1 View and Project matrix

The distribution function discussed earlier first determines the shape and rotation at the origin of the model space using Σ, and then translates it to world space using μ. To render it onto the canvas, it must first be transformed to camera space using a view transformation, and then a projection transformation aligns the perspective space with the pixels for rasterization.

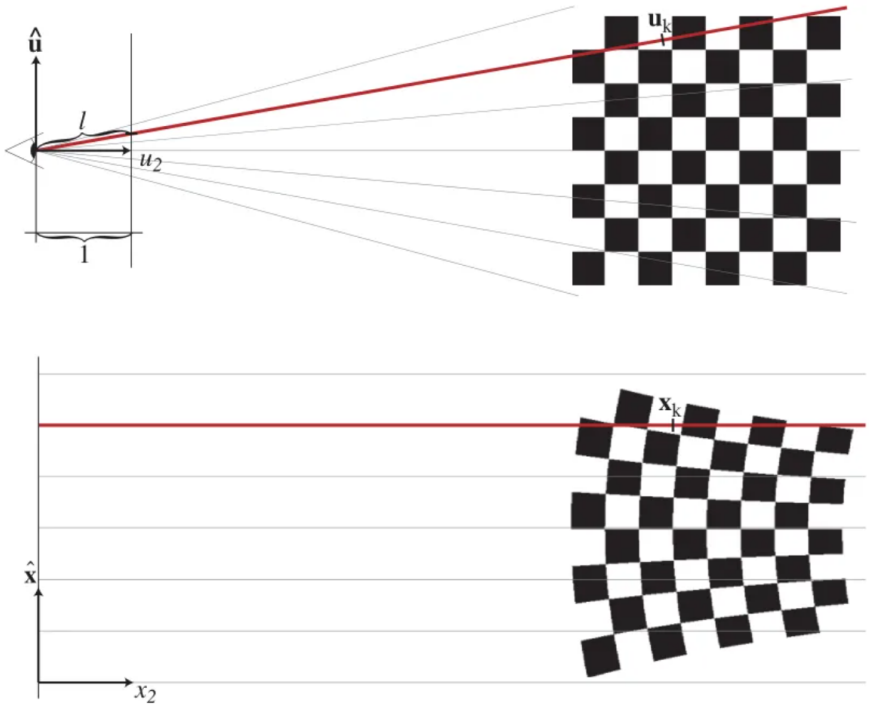

Figure 4: Transforming the volume from camera to ray space. Top: camera space. Bottom: ray space.

The paper aims to maintain the Gaussian distribution during transformations, which poses more constraints compared to simple triangle meshes. The view transformation V primarily involves rotation and translation, which are affine transformations, so there are no issues. However, the projection transformation P is not affine. Therefore, the paper replaces P with J, which is an affine approximation of the projection. The Jacobian matrix of this approximation is used as follows:

$$\Sigma' = JW\Sigma W^T J^T \tag{6}$$

According to EWA volume splatting [4], a local affine approximation can be obtained by taking a second-order Taylor expansion of the projection matrix around a point. More specifically, the Jacobian matrix in this context is the partial derivative of the projection matrix with respect to the point in camera space at this point. The final projection obtained is illustrated in the figure below:

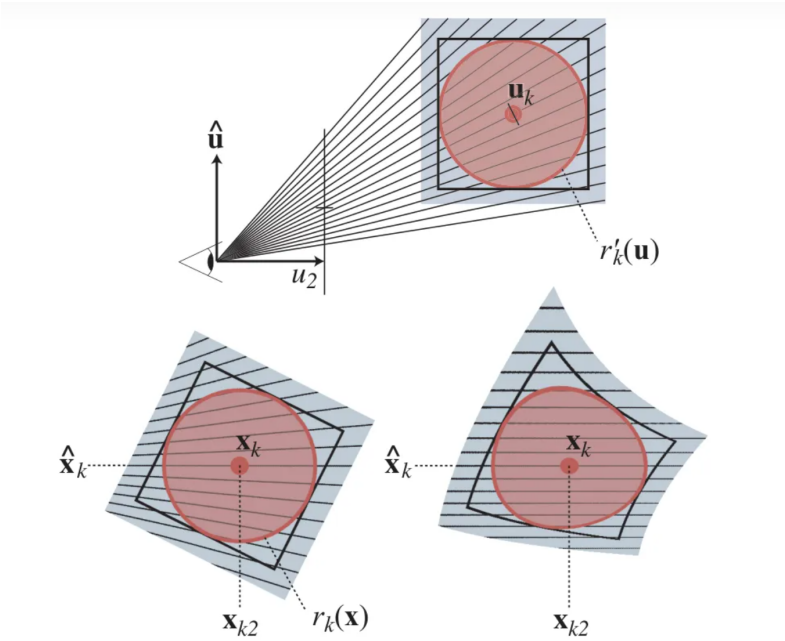

Figure 5: camera space, local affine approximation, and theoretically projected ray space

In the figure, the bottom left shows the approximate result obtained from the paper, while the bottom right shows the ideal project result. Thus, the fundamental unit after transformation remains a 3D Gaussian, with the first two dimensions corresponding to the 2D pixel coordinates xy on the canvas, and the third dimension corresponding to the depth entering from the pixel.

This is the most difficult part in rendering the 3D Gaussian, and my implementation is as following:

```
def computeCov2D(mean, focal_x, focal_y, tan_fovx, tan_fovy, cov3D, viewmatrix):
    # The following models the steps outlined by equations 29
    # and 31 in "EWA Splatting" (Zwicker et al., 2002).
    # Additionally considers aspect / scaling of viewport.
    # Transposes used to account for row-/column-major conventions.

    t = transformPoint4x3(mean, viewmatrix)

    limx = 1.3 * tan_fovx
    limy = 1.3 * tan_fovy
    txtz = t[0] / t[2]
    tytz = t[1] / t[2]
    t[0] = min(limx, max(-limx, txtz)) * t[2]
    t[1] = min(limy, max(-limy, tytz)) * t[2]

    J = np.array(
        [
            [focal_x / t[2], 0, -(focal_x * t[0]) / (t[2] * t[2])],
            [0, focal_y / t[2], -(focal_y * t[1]) / (t[2] * t[2])],
```

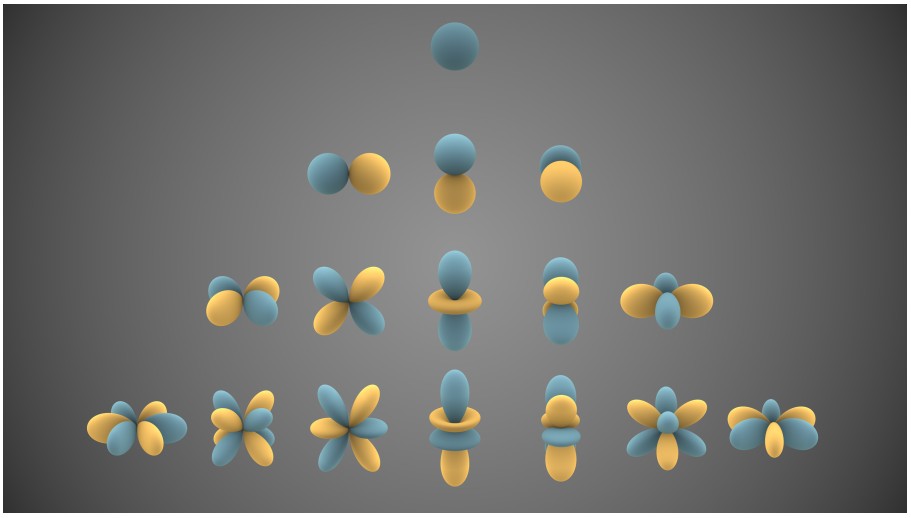

Figure 6: Enter Caption

```
            [0, 0, 0],
        ]
    )
    W = viewmatrix[:3, :3]
    T = np.dot(J, W)

    cov = np.dot(T, cov3D)
    cov = np.dot(cov, T.T)

    # Apply low-pass filter
    # Every Gaussia should be at least one pixel wide/high
    # Discard 3rd row and column
    cov[0, 0] += 0.3
    cov[1, 1] += 0.3
    return [cov[0, 0], cov[0, 1], cov[1, 1]]
```

5.2 Compute color from Spherical Harmonics

In 3D Gaussian Splatting, spherical harmonics are used to represent color, leveraging their mathematical properties to handle directional data efficiently. Spherical harmonics are a set of orthogonal functions defined on the surface of a sphere and are categorized by order (l) and degree (m), each representing different frequency characteristics.

$$f(s) = \sum_{l=0}^{n-1} \sum_{m=-l}^{l} C_l^m Y_l^m(s) \tag{7}$$

The core idea is to represent color, which can vary based on the view direction, using these functions. In practice, each Gaussian splat—a blurred point in 3D space—is associated with directional color information. During the rendering process, the color seen from different angles is sampled. This directional color data is then projected onto the space of spherical harmonics, which involves calculating a set of coefficients. These coefficients essentially describe how the color distribution changes with direction.

Once the spherical harmonics coefficients are computed, they are stored as part of the attributes of the Gaussian splat. When rendering from a specific viewpoint, these coefficients can be used to reconstruct the color information for that direction efficiently.

The code is following:

```
def computeColorFromSH(deg, pos, campos, sh):
    # The implementation is loosely based on code for
    # "Differentiable Point-Based Radiance Fields for
    # Efficient View Synthesis" by Zhang et al. (2022)
```

```
    dir = pos - campos
    dir = dir / np.linalg.norm(dir)

    result = SH_C0 * sh[0]

    if deg > 0:
        x, y, z = dir
        result = result - SH_C1 * y * sh[1] + SH_C1 * z * sh[2] - SH_C1 * x * sh[3]

        if deg > 1:
            xx = x * x
            yy = y * y
            zz = z * z
            xy = x * y
            yz = y * z
            xz = x * z
            result = (
                result
                + SH_C2[0] * xy * sh[4]
                + SH_C2[1] * yz * sh[5]
                + SH_C2[2] * (2.0 * zz - xx - yy) * sh[6]
                + SH_C2[3] * xz * sh[7]
                + SH_C2[4] * (xx - yy) * sh[8]
            )

            if deg > 2:
                result = (
                    result
                    + SH_C3[0] * y * (3.0 * xx - yy) * sh[9]
                    + SH_C3[1] * xy * z * sh[10]
                    + SH_C3[2] * y * (4.0 * zz - xx - yy) * sh[11]
                    + SH_C3[3] * z * (2.0 * zz - 3.0 * xx - 3.0 * yy) * sh[12]
                    + SH_C3[4] * x * (4.0 * zz - xx - yy) * sh[13]
                    + SH_C3[5] * z * (xx - yy) * sh[14]
                    + SH_C3[6] * x * (xx - 3.0 * yy) * sh[15]
                )
    result += 0.5
    return np.clip(result, a_min=0, a_max=1)
```

Finally, the typical point-based rendering equation can be expressed as follows:

$$C = \sum_{i \in N} c_i \alpha_i \prod_{j=1}^{i-1} (1 - \alpha_j) \tag{8}$$

where c_i is the color of each point and α_i is given by evaluating a 2D Gaussian with covariance Σ multiplied with a learned per-point policy. In the origin paper, the authors use a title-based algorithm to leverage existing CPU optimization, and reach a rendering frame rates over 100 FPS. In my own implementation, I use CPU to simulate the basic idea, and just simply blend the colors.

```
    def render(
        self, point_list, W, H, points_xy_image, features, conic_opacity, bg_color
    ):

        out_color = np.zeros((H, W, 3))
        pbar = tqdm(range(H * W))

        # loop pixel
        for i in range(H):
            for j in range(W):
```

```
                pbar.update(1)
                pixf = [i, j]
                C = [0, 0, 0]

                # loop gaussian
                for idx in point_list:

                    # init helper variables, transmirrance
                    T = 1

                    # Resample using conic matrix
                    # (cf. "Surface Splatting" by Zwicker et al., 2001)
                    xy = points_xy_image[idx]  # center of 2d gaussian
                    d = [
                        xy[0] - pixf[0],
                        xy[1] - pixf[1],
                    ]  # distance from center of pixel
                    con_o = conic_opacity[idx]
                    power = (
                        -0.5 * (con_o[0] * d[0] * d[0] + con_o[2] * d[1] * d[1])
                        - con_o[1] * d[0] * d[1]
                    )
                    if power > 0:
                        continue

                    # Eq. (2) from 3D Gaussian splatting paper.
                    # Compute color
                    alpha = min(0.99, con_o[3] * np.exp(power))
                    if alpha < 1 / 255:
                        continue
                    test_T = T * (1 - alpha)
                    if test_T < 0.0001:
                        break

                    # Eq. (3) from 3D Gaussian splatting paper.
                    color = features[idx]
                    for ch in range(3):
                        C[ch] += color[ch] * alpha * T

                    T = test_T

                # get final color
                for ch in range(3):
                    out_color[j, i, ch] = C[ch] + T * bg_color[ch]

        return out_color
```

6 Training and Optimization

In 3D Gaussian Splatting (3DGS), the training process for Gaussian ellipsoids involves several key stages to accurately capture and represent a 3D scene. The process begins with the initialization phase, where a set of Gaussian ellipsoids is positioned within the scene. Each ellipsoid is characterized by parameters such as its position, shape, and color attributes. The shape and orientation of an ellipsoid are determined by its covariance matrix, which dictates how it spreads in different directions. Additionally, color information, potentially including spherical harmonics coefficients for directional variation, is associated with each ellipsoid.

Once the initialization is complete, the optimization phase commences. The objective here is to refine the parameters of the ellipsoids to minimize the difference between the rendered scene and the ground truth data. This data could be in the form of images, point clouds, or other representations of

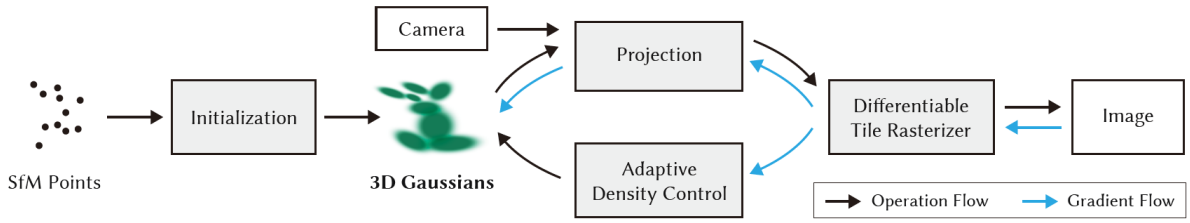

Figure 7: Enter Caption

the 3D environment. The optimization is typically guided by a loss function that quantifies the disparity between the rendered output and the target data. Gradient descent or similar optimization techniques are employed to adjust the parameters iteratively, seeking to minimize the loss function.

As the optimization progresses, various aspects of the ellipsoids are updated. These include their positions, which can shift to better align with the actual structure of the scene, and their sizes and orientations, which are adjusted through changes in the covariance matrix. The color attributes are also refined to ensure that the rendered colors match the observed data as closely as possible. This iterative adjustment is critical for achieving a high-fidelity representation of the scene.

During each iteration of the optimization, the current configuration of Gaussian ellipsoids is rendered. This rendering process involves projecting the ellipsoids onto a 2D plane and calculating their contributions to the final image, including aspects like color and intensity. The rendered image is then compared against the ground truth data, and the loss function is evaluated to guide further adjustments.

The training process continues until convergence, which occurs when the loss function reaches a stable minimum, indicating that the rendered output sufficiently matches the target data. At this point, the Gaussian ellipsoids have been fine-tuned to accurately represent the geometry, texture, and lighting of the scene. This method provides a compact and efficient representation of complex 3D environments, allowing for realistic rendering and analysis.

This is also a big part in 3D Gaussian Splatting, and I am still studying it. So I will not cover this part in my report.

7 Conclusion

That's all for my report on 3D Gaussian Splatting, focusing primarily on how 3D Gaussians represent geometry and how they are transformed onto the screen. The original paper contains many more challenging sections, such as SfM (Structure from Motion) initialization of 3D Gaussian scenes, tile-based sorting and shading, backpropagation gradient optimization, and adaptive adjustment of Gaussian density.

In the filed of computer graphics and computer vision, Gaussian splatting has emerged as one of the hotest topic. It still has limitation in storage requirement, bad quality in dynamic scene, etc. However, the idea of 3D Gaussian Splatting is undeniably a breakthrough and is definitely worth studying for people in related fields.

References

[1] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics*, 42(4), July 2023.

[2] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV*, 2020.

[3] Stoked. Ecstatica — a truly unique survival horror pc game from 1994, 2023.

[4] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Ewa volume splatting. In *Proceedings Visualization, 2001. VIS '01.*, pages 29–538, Oct 2001.