DARB-Splatting: Generalizing Splatting with Decaying Anisotropic Radial Basis Functions

Supplementary Material

1. DARB-Splatting

We generalize the reconstruction kernel to include non-exponential functions by introducing a broader class of Decaying Anisotropic Radial Basis Functions (DARBFs). One of the main reasons why non-exponential functions have not been widely explored is the advantageous integration property of the Gaussian function, which simplifies the computation of the 2D covariance of a splat. However, through Monte Carlo experiments, we demonstrate that DARBFs can also exhibit this desirable property, even though most of them lack a closed-form solution for integration. Additionally, the integration of DARBFs does not generally relate to other DARBFs.

We also explained (Sec. **??**), using an example, that in 3DGS [5], the opacity contribution for each pixel from a reconstruction kernel (3D) is derived from its splats (2D), rather than from their 3D volume. Based on this, we consider the covariance in 3D as a variable representing the 2D covariances in all directions. In the next section, we describe the Monte Carlo experiments we conducted, supported by mathematical equations.

In surface reconstruction tasks [2], DARBFs leverage principal component analysis (PCA) of the local covariance matrix to identify directionally dependent features and orient the 3D ellipsoids accordingly. This approach enables DARBFs to model local anisotropies in the data and reconstruct surfaces, preserving fine details more effectively than isotropic models. Furthermore, the decaying nature of these functions, as presented in Table **??**, results in a more localized influence, effectively focusing the function within a certain radius. This localization is advantageous in 3D reconstruction, where only neighboring points contribute significantly to a given point in space, ensuring smooth blending. Therefore, we can conclude that all DARBFs are suitable for splatting. Although there are many DARBFs, we focus on a selected few here due to limited space. In the following sections, we elaborate on the mathematical formulation of these selected DARBFs, outline their computational implementation, and demonstrate their utility in accurately modeling complex opacity distributions in the context of 3D scene reconstruction.

1.1. Mathematical Expressions of Monte Carlo Experiments

When it comes to 3DGS [5], we initially start with a 3×3 covariance matrix in the 3D world coordinate system. By using,

$$\Sigma' = JW\Sigma W^T J^T \tag{1}$$

we obtain a 3×3 covariance matrix (Σ') in the camera coordinate space. According to the integration property mentioned in the EWA Spatting paper [8], this Σ' can be projected into a 2×2 covariance matrix in the image space ($\Sigma'_{2\times 2}$) by simply removing the third row and column of Σ'. However, the same process does not apply to other DARBFs. For instance, there is no closed-form solution for the marginal integration of the half-cosine and raised-cosine functions used in this paper. To simplify the understanding of this integration process and address this issue, we conducted the following experiment.

Experimental Setup. First, we introduce our 3D point space with x, y, z coordinates in equally spaced intervals for N number of points, a random mean vector (μ) and a random 3×3 covariance matrix (Σ). Based on a predetermined limit specific for each DARBF (we will discuss about this in Sec. 1.2), we calculate the density/power assigned by the DARB kernel at a particular point $\boldsymbol{x}$ in 3D space as follows:

$$P = \cos\left(\frac{2\pi\left(\boldsymbol{x}-\boldsymbol{\mu}\right)^T\left(\Sigma\right)^{-1}\left(\boldsymbol{x}-\boldsymbol{\mu}\right)}{36}\right), \tag{2}$$

where $\boldsymbol{x} = \begin{bmatrix} x_i & y_i & z_i \end{bmatrix}$. As for the integration, we take the sum of these P ($P \in \mathbb{R}^{N\times N\times N}$) matrices along one dimension (for instance, along z axis) and name it $total\_density$, which can be obtained as follows:

$$\text{total\_density}\,(x, y) = \sum_z P(x, y, z) \tag{3}$$

where $total\_density \in \mathbb{R}^{N\times N}$. This will, for example, integrate the cosine kernel in Eq. 2 along z-direction, collapsing into a 2D density in XY plane. Following this integration, these total densities will be normalized as follows:

$$\text{total\_density}_{\text{normalized}} = \frac{\text{total\_density}}{\max(\text{total\_density})} \tag{4}$$

This normalization step does not change the typical covariance relationship. To compare with other functions' projection better, we use this normalization, so that the maximum of total density will be equal one.

For the visualization of these 2D densities, we create a 2D mesh grid by using only x, y coordinate matrices called $coords \in \mathbb{R}^{N^2\times 2}$. At the same time, we flatten the 2D density matrix and get a density grid as $density \in \mathbb{R}^{N^2\times 1}$.

If the dimensions of x, y coordinates are different, we need to repeat the $coords$ and $density$ arrays separately to perfectly align each 2D coordinate for its corresponding density value. Since we use the same dimension for x, y coordinates, we can skip this step. By using these $coords$ and $density$ matrices, we then calculate the weighted covariance matrix as follows:

$$\bar{x} = \frac{\sum_{i=1}^{N'} w_i x_i}{\sum_{i=1}^{N'} w_i} \quad \bar{y} = \frac{\sum_{i=1}^{N'} w_i y_i}{\sum_{i=1}^{N'} w_i}$$

where $N' = N^2$ and w_i denote the corresponding parameters from the $density$ matrix. Finally, the vector $\bar{\boldsymbol{m}}$ is given by:

$$\bar{\boldsymbol{m}} = \begin{bmatrix} \bar{x} \\ \bar{y} \end{bmatrix}_{2\times 1} \tag{5}$$

By using the above results, we can determine the projected 2×2 covariance matrix ($\Sigma'_{2\times2}$) as follows:

$$\Sigma'_{2\times2} = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} \\ \sigma_{xy} & \sigma_{yy} \end{bmatrix}$$

where σ_{xx}, σ_{yy} and σ_{xy} terms can be determined as:

$$\sigma_{xx} = \frac{\sum_{i=1}^{N} w_i (x_i - \bar{x})^2}{\sum_{i=1}^{N} w_i} \qquad \sigma_{yy} = \frac{\sum_{i=1}^{N} w_i (y_i - \bar{y})^2}{\sum_{i=1}^{N} w_i}$$

$$\sigma_{xy} = \frac{\sum_{i=1}^{N} w_i (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{N} w_i}$$

We can express this entire operation in matrix form as follows:

$$\Sigma'_{2\times2} = \frac{1}{\sum_{i=1}^{N'} w_i} \sum_{i=1}^{N'} w_i \left(\boldsymbol{x}_i - \bar{\boldsymbol{m}}\right) \left(\boldsymbol{x}_i - \bar{\boldsymbol{m}}\right)^T \tag{6}$$

$$\text{where} \quad \boldsymbol{x}_i = \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

Based on this simulation, we received the projected 2×2 covariance matrix ($\Sigma'_{2\times2}$) for different DARBFs and identified that they are not integrable for volume rendering [8]. Simply saying, we cannot directly obtain the first two rows and columns of $\Sigma'_{2\times2}$ from the first two rows and first tow columns of the 3D covariance matrix $\Sigma'_{3\times3}$ as they are. But we noticed that there is a common ratio between the values of these two matrices. To resolve this issue, we introduce a correction factor ψ as a scalar to multiply with the $\Sigma'_{2\times2}$ matrix. This scalar holds different values for different DARBFs since their density kernels act differently.

1.2. Determining the Boundaries of DARBFs

From the 1D signal reconstruction simulations (Sec. **??** and Sec. 4) and the splatting results (Sec. 4), we demonstrate that strictly decaying functions can represent the scenes better. Therefore, we use the limits for each function to ensure the strictly decaying nature, while also considering the size of each splat. By restricting to a single pulse, we achieve a more localized representation, resulting in better quality. Incorporating more pulses allows them to cover a larger region compared to one pulse (with the same ξ and β), leading to memory reduction, albeit with a tradeoff in quality.

In 3DGS [5], opacity modeling happens after the projection of the 3D covariance (Σ) in world coordinate space onto 2D covariance ($\Sigma'_{2\times2}$) in image space. By using the inverse covariance ($\Sigma'^{-1}_{2\times2}$) and the difference between the center of the splat (μ') and the coordinates of the selected pixel, they introduce the Mahalnobis component (Eq. **??**) within the Gaussian kernel to model the opacity distribution across each splat. When determining the final color of a particular pixel, they have incorporated a bounding box mechanism to identify the area which a splat can have the effect when modeling the opacity, so that they can do the tile-based rasterization using the computational resources efficiently.

Since the Gaussian only has a main lobe, we can simply model the opacity distribution across a splat using the bounding box mentioned in Sec. **??**. This bounding box, determined by the radius $R = 3 \cdot \sqrt{\max\{\lambda_1, \lambda_2\}}$, where λ_1 and λ_2 denote the eigenvalues of the 2D covariance matrix $\Sigma'_{2\times2}$ (Sec. **??**), will cover most of the function (main lobe), affecting the opacity modeling significantly. However, in our DARBFs, we have multiple side lobes which can have an undesirable effect on this bounding box unless the range is specified correctly. If these side lobes are included within the bounding box, each splat will have a ring effect in their opacity distributions.

To avoid this ring effect, we identified the range of the horizontal spread of the main lobe of each DARBF in terms of their Mahalanobis distance component and introduced a limit in opacity distribution to carefully remove the effects from their side lobes. In our Monte Carlo experiments (Sec. 1.1), we used this limit in the 3D DARB kernel, as we directly perform the density calculation in 3D and the projection onto 2D image space afterwards. For example, let us consider the 3D raised cosine as follows:

$$w = 0.5 + 0.5 \cos\left(\frac{2\pi d_M^2}{5}\right) \tag{7}$$

where $\xi = \frac{5}{2\pi}$ and $\beta = 2$ according to the standard expression mentioned in Table **??**. To avoid the side lobes and only use the main lobe, we assess the necessary range that we should consider with the Gaussian curves in 1D and chose the following limit (according to Table **??**):

$$d_M^2 < 6.25 = \left(\pi \times \frac{5}{2\pi}\right)^2 \tag{8}$$

If the above limit is not satisfied by the Mahalanobis component, the density value will be taken as zero for those cases. As in the Table **??**, this limit will be different for different DARBFs since each DARBF shows different characteristics regarding their spread, main lobe and side lobes. Applying these limits will help to consider the 100% support of the main lobe of each DARBF into the bounding box.

Even though we apply these limits on the 3D representation of each kernel and calculate the densities, in our reconstruction pipeline, we use these limits on DARB splats (2D) similar to 3DGS [5]. In our experiments, our main target was to identify the relationship between the 2D covariance $\Sigma'_{2\times2}$ and the 3D submatrix $\Sigma'_{3\times3}$ (Sec. 1.1), and implement these limits on 3D DARB kernels.

2. DARB-Splatting Implementation

Here, we present the reconstruction kernel (3D) and splat (2D) functions (footprint functions) for selected DARBFs, along with their respective derivative term modifications related to backpropagation, in both mathematical expressions and CUDA codes. These modifications have been incorporated into the splatting pipeline and CUDA code changes. In the code, d denotes $x - \boldsymbol{\mu}'$, and $con = \begin{bmatrix} a & b \\ b & c \end{bmatrix}$ denotes the inverse of the 2D covariance matrix $(\Sigma'_{2\times2})^{-1}$ in the mathematical form. For each DARBF, we clearly show ξ and d_M in the code. We use a unique correction factor ψ for each DARBF, determined through Monte Carlo experiments, to compute $\Sigma'_{2\times2}$ from Σ'.

2.1. Raised Cosine Splatting

Here, a single pulse of the raised cosine signal is selected for enhanced performance. The 3D raised cosine function is as follows:

$$0.5 + 0.5 \cos\left(\frac{1}{\xi}\left(d_M\right)\right), \quad \frac{1}{\xi}\left(d_M\right) \leq \pi \tag{9}$$

The raised cosine splat function is as follows:

$$w = 0.5 + 0.5 \cos\left(\frac{\sqrt{\left(x-\boldsymbol{\mu}'\right)^T \boldsymbol{\Sigma}'^{-1}_{2\times 2}\left(x-\boldsymbol{\mu}'\right)}}{\xi}\right), \tag{10}$$

$$\frac{\sqrt{\left(x-\boldsymbol{\mu}'\right)^T \boldsymbol{\Sigma}'^{-1}_{2\times 2}\left(x-\boldsymbol{\mu}'\right)}}{\xi} \leq \pi$$

Modifications in derivative terms related to the raised cosine splat during backpropagation are provided next.

$$\frac{\partial w}{\partial(x-\boldsymbol{\mu}')} = -\frac{0.5}{\xi}\sin\left(\frac{\sqrt{\left(x-\boldsymbol{\mu}'\right)^T \boldsymbol{\Sigma}'^{-1}_{2\times 2}\left(x-\boldsymbol{\mu}'\right)}}{\xi}\right)\cdot\frac{\boldsymbol{\Sigma}'^{-1}_{2\times 2}(x-\boldsymbol{\mu}')}{\sqrt{\left(x-\boldsymbol{\mu}'\right)^T \boldsymbol{\Sigma}'^{-1}_{2\times 2}\left(x-\boldsymbol{\mu}'\right)}} \tag{11}$$

$$\frac{\partial w}{\partial \boldsymbol{\Sigma}'^{-1}_{2\times 2}} = -\frac{0.5}{\xi}\sin\left(\frac{\sqrt{\left(x-\boldsymbol{\mu}'\right)^T \boldsymbol{\Sigma}'^{-1}_{2\times 2}\left(x-\boldsymbol{\mu}'\right)}}{\xi}\right)\cdot\frac{\left(x-\boldsymbol{\mu}'\right)\left(x-\boldsymbol{\mu}'\right)^T}{\sqrt{\left(x-\boldsymbol{\mu}'\right)^T \boldsymbol{\Sigma}'^{-1}_{2\times 2}\left(x-\boldsymbol{\mu}'\right)}} \tag{12}$$

The following CUDA code modifications were implemented to support raised cosine splatting.

```
float correctionFactor =  0.655f; // The correction factor we introduce
cov *= correctionFactor ; // Multiply the 3D covariance matrix by the correction factor

float ellipse = con_o.x * d.x * d.x + con_o.z * d.y * d.y + 2.f * con_o.y * d.x * d.y; // dM**2
if (ellipse < 0 || ellipse > 6.25f) // Limit set on 2D ellipse to restrict to one pulse
    continue;
float ellipse_root = sqrt(ellipse); // dM
float ellipseFactor =  M_PI * ellipse_root / 2.5f; // Here scaling factor, xi = 5/(2 * math_pi)
const float Cosine = 0.5f + .5f * cos(ellipseFactor);
const float alpha = min(0.99f, con_o.w * Cosine); // w * raised cosine
```

Listing 1. Modifications in forward propagation in CUDA rasterizer for raised cosine splatting.

```
float correctionFactor = 0.655f;
cov2D *= correctionFactor;

// Gradients of loss w.r.t. entries of 2D covariance matrix,
// given gradients of loss w.r.t. conic matrix (inverse covariance matrix).
// e.g., dL / da = dL / d_conic_a * d_conic_a / d_a
dL_da = correctionFactor * denom2inv * (-c * c * dL_dconic.x + 2 * b * c * dL_dconic.y + (denom - a * c) * dL_dconic.z);
dL_dc = correctionFactor *denom2inv * (-a * a * dL_dconic.z + 2 * a * b * dL_dconic.y + (denom - a * c) * dL_dconic.x);
dL_db = correctionFactor *denom2inv * 2 * (b * c * dL_dconic.x - (denom + 2 * b * b) * dL_dconic.y + a * b * dL_dconic.z);

float ellipse = (con_o.x * d.x * d.x + con_o.z * d.y * d.y + 2.f * con_o.y * d.x * d.y);
if (ellipse < 0 || ellipse > 6.25f) // limit set on 2D ellipse to restrict to one pulse
    continue;
float ellipse_root = sqrt(ellipse);   // dM
float ellipseFactor =  M_PI *ellipse_root / 2.5f;
const float Cosine = .5f + .5f* cos(ellipseFactor); // raised cosine
const float alpha = min(0.99f, con_o.w * Cosine); // w * raised cosine

// Helpful reusable temporary variables
const float dL_dCosine = con_o.w * dL_dalpha;
const float sindL_dCosine = sin(ellipseFactor)* dL_dCosine;
const float factor1 =  - M_PI * sindL_dCosine /5.f / ellipse_root;
const float factor2 = factor1 * 0.5 ;
const float dL_ddelx = ( con_o.x * d.x + con_o.y * d.y)* factor1;
const float dL_ddely = ( con_o.z * d.y + con_o.y * d.x)* factor1;

atomicAdd(&dL_dmean2D[global_id].x, dL_ddelx * ddelx_dx);
atomicAdd(&dL_dmean2D[global_id].y, dL_ddely * ddely_dy);
// Update gradients w.r.t. 2D covariance (2x2 matrix, symmetric)
atomicAdd(&dL_dconic2D[global_id].x, d.x * d.x * factor2);
atomicAdd(&dL_dconic2D[global_id].y, d.x * d.y * factor2);
atomicAdd(&dL_dconic2D[global_id].w, d.y * d.y * factor2);
// Update gradients w.r.t. opacity of the Cosine
atomicAdd(&(dL_dopacity[global_id]), Cosine * dL_dalpha);
```

Listing 2. Modifications in backward propagation in CUDA rasterizer for raised cosine splatting.

2.2. Half-cosine Squared Splatting

The 3D half-cosine square function we selected is as follows:

$$\cos\left(\frac{1}{\xi}\left(d_M\right)^2\right), \quad \frac{1}{\xi}\left(d_M\right)^2 \leq \frac{\pi}{2}$$

The corresponding half-cosine squared splat function is as follows:

$$w = \cos\left(\frac{\left(x-\boldsymbol{\mu}'\right)^T \left(\Sigma'_{2\times 2}\right)^{-1}\left(x-\boldsymbol{\mu}'\right)}{\xi}\right), \tag{13}$$

$$\frac{\sqrt{\left(x-\boldsymbol{\mu}'\right)^T \boldsymbol{\Sigma}'^{-1}_{2\times 2}\left(x-\boldsymbol{\mu}'\right)}}{\xi} \leq \frac{\pi}{2}$$

Adjustments to the derivative terms associated with the half-cosine squared splat during backpropagation are given below.

$$\frac{dw}{d(x-\boldsymbol{\mu}')} = \frac{-2\left(\Sigma'_{2\times 2}\right)^{-1}\left(x-\mu'\right)\,\sin\left(\frac{(x-\boldsymbol{\mu}')^T(\Sigma'_{2\times 2})^{-1}(x-\boldsymbol{\mu}')}{\xi}\right)}{\xi} \tag{14}$$

$$\frac{dw}{d((\Sigma'_{2\times 2})^{-1})} = \frac{-(x-\mu')\,\left(x-\boldsymbol{\mu}'\right)^T\,\sin\left(\frac{(x-\boldsymbol{\mu}')^T(\Sigma'_{2\times 2})^{-1}(x-\boldsymbol{\mu}')}{\xi}\right)}{\xi} \tag{15}$$

The following CUDA code changes were made to support half-cosine squared splatting.

```
float correctionFactor =  1.36f;  // The correction factor we introduce
cov *= correctionFactor ; // Multiply the 3D covariance matrix by the correction factor
```

```
float ellipse = con_o.x * d.x * d.x + con_o.z * d.y * d.y + 2.f * con_o.y * d.x * d.y; // dm_squared
if (ellipse < 0.f ||  ellipse > 9.f ) // limit set on 2D ellipse to restrict half cosine pulse
    continue;
float ellipseFactor =   0.175f *ellipse; // 2 * M_PI * ellipse / 36;
const float Cosine = cos(ellipseFactor);
const float alpha = min(0.99f, con_o.w * Cosine); // w * cosine
```

Listing 3. Modifications in forward propagation in CUDA rasterizer for half-cosine squared splatting.

```
float correctionFactor = 1.36f;
cov2D *= correctionFactor;

// Gradients of loss w.r.t. entries of 2D covariance matrix,
// given gradients of loss w.r.t. conic matrix (inverse covariance matrix).
// e.g., dL / da = dL / d_conic_a * d_conic_a / d_a
dL_da = correctionFactor * denom2inv * (-c * c * dL_dconic.x + 2 * b * c * dL_dconic.y + (denom - a * c) * dL_dconic.z);
dL_dc = correctionFactor *denom2inv * (-a * a * dL_dconic.z + 2 * a * b * dL_dconic.y + (denom - a * c) * dL_dconic.x);
dL_db = correctionFactor *denom2inv * 2 * (b * c * dL_dconic.x - (denom + 2 * b * b) * dL_dconic.y + a * b * dL_dconic.z);

float ellipse = con_o.x * d.x * d.x + con_o.z * d.y * d.y + 2.f * con_o.y * d.x * d.y;
if (ellipse < 0 || ellipse > 9.f) // limit set on 2D ellipse
    continue;
float ellipseFactor = 0.174f *ellipse ; // 2* M_PI * ellipse / 36.f;
const float Cosine = cos(ellipseFactor); // half cosine square
const float alpha = min(0.99f, con_o.w * Cosine);

// Helpful reusable temporary variables
const float dL_dCosine = con_o.w * dL_dalpha;
const float sindL_dCosine = -0.349* sin(ellipseFactor) * dL_dCosine;
const float halfsindL_dCosine = 0.5f * sindL_dCosine;
const float dCosine_ddelx =  ( con_o.x * d.x + con_o.y * d.y)* sindL_dCosine ;
const float dCosine_ddely = ( con_o.z * d.y + con_o.y * d.x)* sindL_dCosine;

// Update gradients w.r.t. 2D mean position of the half cosine squared
atomicAdd(&dL_dmean2D[global_id].x,  dCosine_ddelx * ddelx_dx);
atomicAdd(&dL_dmean2D[global_id].y,  dCosine_ddely * ddely_dy);
// Update gradients w.r.t. 2D covariance (2x2 matrix, symmetric)
atomicAdd(&dL_dconic2D[global_id].x,  d.x * d.x * halfsindL_dCosine);
atomicAdd(&dL_dconic2D[global_id].y, d.x * d.y * halfsindL_dCosine );
atomicAdd(&dL_dconic2D[global_id].w,  d.y * d.y * halfsindL_dCosine);
// Update gradients w.r.t. opacity of the half cosine squared
atomicAdd(&(dL_dopacity[global_id]), Cosine * dL_dalpha);
```

Listing 4. Modifications in backward propagation in CUDA rasterizer for half-cosine squared splatting.

2.3. Sinc Splatting

Here, the sinc function refers to a single pulse of the modulus sinc function. This configuration was selected due to improved performance. The corresponding 3D sinc function is provided below:

$$\left|\frac{\sin\left(\frac{1}{\xi}\left(d_M\right)\right)}{\frac{1}{\xi}\left(d_M\right)}\right|, \quad \frac{1}{\xi}\left(d_M\right) \leq \pi$$

The related sinc splat function is as follows:

$$w = \left|\frac{\sin\left(\frac{\sqrt{(x-\boldsymbol{\mu}')^T(\Sigma'_{2\times 2})^{-1}(x-\boldsymbol{\mu}')}}{\xi}\right)}{\left(\frac{\sqrt{(x-\boldsymbol{\mu}')^T(\Sigma'_{2\times 2})^{-1}(x-\boldsymbol{\mu}')}}{\xi}\right)}\right|, \tag{16}$$

$$\frac{\sqrt{(x-\boldsymbol{\mu}')^T\,\boldsymbol{\Sigma}'^{-1}_{2\times 2}\,(x-\boldsymbol{\mu}')}}{\xi} \leq \pi$$

The modifications to the derivative terms related to the sinc splat in backpropagation are outlined below.

$$\frac{\partial w}{\partial(x-\boldsymbol{\mu}')} = \operatorname{sgn}\left(\frac{\sin(A)}{A}\right)\cdot\frac{A\cos(A)-\sin(A)}{A^2}\cdot\frac{2(\Sigma'_{2\times 2})^{-1}(x-\mu')}{\xi}, \tag{17}$$

$$\frac{\partial w}{\partial(\Sigma'_{2\times 2})^{-1}} = \operatorname{sgn}\left(\frac{\sin(A)}{A}\right)\cdot\frac{A\cos(A)-\sin(A)}{A^2}\cdot\frac{(x-\mu')(x-\mu')^T}{\xi}, \tag{18}$$

where $\quad A = \frac{(x-\mu')^T(\Sigma'_{2\times 2})^{-1}(x-\mu')}{\xi}$.

The following CUDA code changes were made to support the sinc splatting described here.

```
float correctionFactor =  1.18f; // The correction factor we introduce
cov *= correctionFactor ; // Multiply the 3D covariance matrix by the correction factor

float constA = 3.0f/ M_PI;
float ellipse = con_o.x * d.x * d.x + con_o.z * d.y * d.y + 2.f * con_o.y * d.x * d.y; // dM**2
if (ellipse <=  0 || ellipse > 9.f) // limit set on 2D ellipse to restrict to one pulse
    continue;
float ellipse_root = sqrt(ellipse); // dM
float ellipse_rootdivA = ellipse_root / constA;
float alpha = min(0.99f, con_o.w * fabs(sin(ellipse_rootdivA) / ellipse_rootdivA)); // w * modulus sinc
```

Listing 5. Modifications in forward propagation in CUDA rasterizer for sinc splatting.

```
float correctionFactor = 1.18f;
cov2D *= correctionFactor;

// Gradients of loss w.r.t. entries of 2D covariance matrix,
// given gradients of loss w.r.t. conic matrix (inverse covariance matrix).
// e.g., dL / da = dL / d_conic_a * d_conic_a / d_a
```

```
dL_da = correctionFactor * denom2inv * (-c * c * dL_dconic.x + 2 * b * c * dL_dconic.y + (denom - a * c) * dL_dconic.z);
dL_dc = correctionFactor *denom2inv * (-a * a * dL_dconic.z + 2 * a * b * dL_dconic.y + (denom - a * c) * dL_dconic.x);
dL_db = correctionFactor *denom2inv * 2 * (b * c * dL_dconic.x - (denom + 2 * b * b) * dL_dconic.y + a * b * dL_dconic.z);

float constA =3.0f/ M_PI;
float ellipse = con_o.x * d.x * d.x + con_o.z * d.y * d.y + 2.f * con_o.y * d.x * d.y;
if (ellipse <=  0 || ellipse > 9.f ) // limit set on 2D ellipse to restrict to one pulse
    continue;
float ellipse_root = sqrt(ellipse);  // dM
float ellipse_rootdivA = ellipse_root / constA;
const float sinellipse_rootdivA = sin(ellipse_rootdivA);
const float G =  fabs(sinellipse_rootdivA / ellipse_rootdivA); // modulus sinc
const float alpha = min(0.99f, con_o.w * G); // w * modulus sinc

// Helpful reusable temporary variables
const float dL_dG = con_o.w * dL_dalpha;
// Original equation:
// cospart = cos(ellipse_rootdivA) * sin(ellipse_rootdivA) * fabs(1 / ellipse_root) / ellipse_root / fabs(sin(ellipse_rootdivA))
// Simplified equation using copysignf for clarity:
const float cospart = cos(ellipse_rootdivA ) * copysignf(1.0f, sinellipse_rootdivA) / ellipse ;
const float sinpart = constA *fabs(sinellipse_rootdivA) / ellipse / ellipse_root  ;
const float commonpart = (cospart - sinpart)* dL_dG;
const float commonpartdiv2 = commonpart  * 0.5f;
const float dG_ddelx = commonpart * (d.x * con_o.x + d.y * con_o.y);
const float dG_ddely = commonpart * (d.x * con_o.y + d.y * con_o.z);
// Update gradients w.r.t. 2D mean position of the modulus sinc
atomicAdd(&dL_dmean2D[global_id].x, dG_ddelx * ddelx_dx);
atomicAdd(&dL_dmean2D[global_id].y,  dG_ddely * ddely_dy);
// Update gradients w.r.t. 2D covariance (2x2 matrix, symmetric)
atomicAdd(&dL_dconic2D[global_id].x, commonpartdiv2 * d.x * d.x );
atomicAdd(&dL_dconic2D[global_id].y, commonpartdiv2 * d.x * d.y );
atomicAdd(&dL_dconic2D[global_id].w, commonpartdiv2 * d.y * d.y );
// Update gradients w.r.t. opacity of the modulus sinc
atomicAdd(&(dL_dopacity[global_id]), G * dL_dalpha);
```

Listing 6. Modifications in backward propagation in CUDA rasterizer for sinc splatting.

2.4. Inverse Quadratic Splatting

Here, we define the inverse quadratic formulation based on the following 3D inverse quadratic function:

$$\frac{1}{\left[\frac{1}{\xi}\left(d_M\right)^2+1\right]}, \quad d_M \geq 0$$

The corresponding inverse quadratic splat function is as follows:

$$w=\frac{1}{\left[\frac{1}{\xi}\left(x-\boldsymbol{\mu}^{\prime}\right)^T\left(\Sigma_{2\times 2}^{\prime}\right)^{-1}\left(x-\boldsymbol{\mu}^{\prime}\right)+1\right]} \tag{19}$$

The changes done to the derivative terms related to the inverse quadratic splat during backpropagation are detailed below.

$$\frac{\partial w}{\partial z}=-\frac{2}{\xi\left(\frac{1}{\xi} z^T(\Sigma_{2\times 2})^{-1} z+1\right)^2}(\Sigma_{2\times 2})^{-1} z. \tag{20}$$

$$\frac{\partial w}{\partial(\Sigma_{2\times 2})^{-1}}=-\frac{1}{\left(\frac{1}{\xi} z^T(\Sigma_{2\times 2})^{-1} z+1\right)^2} \cdot \frac{1}{\xi} z z^T. \tag{21}$$

where $z=(x-\mu^{\prime})$.

The CUDA code modifications provided below were implemented to enable the inverse quadratic splatting described in this section.

```
float correctionFactor =  1.38f; // The correction factor we introduce
cov *= correctionFactor ; // Multiply the 3D covariance matrix by the correction factor

float ellipse = con_o.x * d.x * d.x + con_o.z * d.y * d.y + 2.f * con_o.y * d.x * d.y ; // dm**2
if (ellipse <=  0)
    continue;
if (ellipse  >= 9.f) // limit set on 2D ellipse
    continue;
float alpha = min(0.99f, con_o.w * (1.f / (ellipse + 1.f))); // w * inverse quadric
```

Listing 7. Modifications in forward propagation in CUDA rasterizer for inverse quadratic splatting.

```
float correctionFactor = 1.38f;
cov2D *= correctionFactor;

// Gradients of loss w.r.t. entries of 2D covariance matrix,
// given gradients of loss w.r.t. conic matrix (inverse covariance matrix).
// e.g., dL / da = dL / d_conic_a * d_conic_a / d_a
dL_da = correctionFactor * denom2inv * (-c * c * dL_dconic.x + 2 * b * c * dL_dconic.y + (denom - a * c) * dL_dconic.z);
dL_dc = correctionFactor *denom2inv * (-a * a * dL_dconic.z + 2 * a * b * dL_dconic.y + (denom - a * c) * dL_dconic.x);
dL_db = correctionFactor *denom2inv * 2 * (b * c * dL_dconic.x - (denom + 2 * b * b) * dL_dconic.y + a * b * dL_dconic.z);

float ellipse = con_o.x * d.x * d.x + con_o.z * d.y * d.y + 2.f * con_o.y * d.x * d.y ;
if (ellipse <=  0)
    continue;
if (ellipse >= 9.f) // limit set on 2D ellipse
    continue;
const float G = (1.f / (ellipse + 1.f)); // inverse quadric
const float alpha = min(0.99f, con_o.w * G);

// Helpful reusable temporary variables
const float dL_dG = con_o.w * dL_dalpha;
const float ellipsepow =  - dL_dG * 2.f / pow(ellipse + 1.f, 2.f);
const float halfellipsepow  = 0.5f * ellipsepow;
const float dG_ddelx = ellipsepow * (d.x * con_o.x + d.y * con_o.y);
const float dG_ddely = ellipsepow * (d.x * con_o.y + d.y * con_o.z);
// Update gradients w.r.t. 2D mean position of the IQF
atomicAdd(&dL_dmean2D[global_id].x,  dG_ddelx * ddelx_dx);
atomicAdd(&dL_dmean2D[global_id].y,  dG_ddely * ddely_dy);
// Update gradients w.r.t. 2D covariance (2x2 matrix, symmetric)
```

```
atomicAdd(&dL_dconic2D[global_id].x, halfellipsepow * d.
     x * d.x );
atomicAdd(&dL_dconic2D[global_id].y, halfellipsepow * d.
     x * d.y );
atomicAdd(&dL_dconic2D[global_id].w, halfellipsepow * d.
     y * d.y );
// Update gradients w.r.t. opacity of the IQF
atomicAdd(&(dL_dopacity[global_id]), G * dL_dalpha);
```

Listing 8. Modifications in backward propagation in CUDA rasterizer for inverse quadratic splatting.

3. Utility Applications of DARB-Splatting

3.1. Enhanced Quality

In terms of splatting, despite Gaussians providing SOTA quality, we demonstrate that the raised cosine function can deliver modestly improved visual quality compared to Gaussians. Our 1D simulations, presented in Sec. **??** and Sec. 4, and the qualitative visual comparisons demonstrated in Fig. 1, illustrate this effectively.

Across the selected DARBFs, only the raised cosine outperforms the Gaussian in terms of quality, albeit by a small margin. The others fail to surpass the Gaussian in terms of quality. The primary reason is that exponentially decaying functions ensure faster blending compared to relatively flatter functions. However, these functions have other utilities, which we will discuss next.

3.2. Reduced Training Time

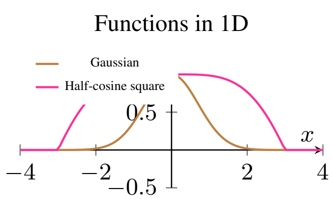

Figure 2. Comparison of Gaussian and $\cos\left(\frac{2\pi x^2}{36}\right)$ functions within the range $|x| \leq 3$ (same variance for both).

According to Fig. 2, which shows the half-cosine square with $\beta = 2$ and $\xi = 36$, along with the Gaussian 1D plot, we can observe that for a single primitive with the same variance, the cosine function can provide higher opacity values. Instead of requiring multiple splats to composite to determine the final pixel color, the cosine function can achieve the same accumulated value required with fewer primitives. Although the cosine function's computation is more time-intensive compared to the Gaussian calculations, the overall training time is reduced due to the lower number of primitives required.

This is further illustrated in Figures 3 and 4, which provide a detailed analysis of the training loss and speed curves across various dataset scenes. Overall, despite having similar training loss curves with 3DGS, half-cosine splatting demonstrates superior performance compared to Gaussians.

3.3. Reduced Memory Usage

As previously mentioned, half cosine squared splatting specifically requires fewer primitives compared to Gaussians. This results in lower memory usage, as they provide higher opacity values across most of the regions they cover. In contrast, Gaussians require more primitives to achieve a similar accumulated opacity coverage. By using fewer half cosine squared primitives, we can achieve the desired color representation in the image space more efficiently. Similarly, sinc splatting and inverse quadratic splatting also consume lesser memory compared to Gaussians. The results in Table 1 and Table 2 further showcase this.

4. Extended Results and Simulations

Extended results. As mentioned in our paper, we trained our models on a single NVIDIA GeForce RTX 4090 GPU and recorded the training time. Since the benchmark models from other papers [3, 5] were trained on different GPUs, we applied a scaling factor to ensure a fair comparison of training times with the *original* papers' results. According to [7], the relative training throughput of the RTX 4090 GPU and other GPU models (specifically, the RTX 3090 and RTX A6000 GPUs) can be determined with respect to a 1xLambdaCloud V100 16GB GPU. By dividing the training time data by these values, we have presented our training time results in a fair and comparable manner in Table **??**.

A detailed breakdown of our results across every scene in the Mip-NeRF 360 [1], Tanks&Temples [6], and Deep Blending [4] datasets, along with their average values per dataset, is provided in Table 1 and Table 2. These results pertain to the selected DARB-Splatting algorithms, namely raised cosine splatting (3DRCS), half-cosine squared splatting (3DHCS), sinc splatting (3DSS), and inverse quadratic splatting (3DIQS). Key evaluation metrics, including PSNR, SSIM, LPIPS, memory usage, and training time for both 7k and 30k iterations, are analyzed in detail. These are presented alongside the results from implementing the *updated codebase* of 3DGS on our single NVIDIA GeForce RTX 4090 GPU to ensure a fair comparison. Our pipeline is anchored on this *updated codebase*, which produces improved results compared to those reported in the original 3DGS paper [5]. As shown in the tables, each DARB-Splatting algorithm demonstrates unique advantages in different utilities.

1D Simulations. In Figures 5, 6, 7, 8, 9, 10, 11, we show an extended version of the initial 1D simulation described in Sec. **??** in our paper. Here, we conduct experiments with various reconstruction kernels in 1D as a toy experiment to understand their signal reconstruction properties. These kernels include Gaussians, cosines, squared cosines, raised cosines, squared raised cosines, and modulus sincs. The reconstruction process is optimized using backpropagation, with different means and variances applied to each kernel. This approach is used to reconstruct various complex signal types, including a square pulse, a triangular pulse, a Gaussian pulse, a half-sinusoid single pulse, a sharp exponential pulse, a parabolic pulse, and a trapezoidal pulse.

We are grateful to the authors of GES [3] for open-sourcing their 1D simulation codes, which we have improved upon for this purpose. Expanding beyond GES, here, we also demonstrate the reconstruction of non-symmetric 1D signals to better represent real-world 3D reconstructions and further explore the capabilities of various DARBFs. As shown in the simulations, Gaussians are not the only effective interpolators; other DARBFs can provide improved 1D signal reconstructions in specific cases.

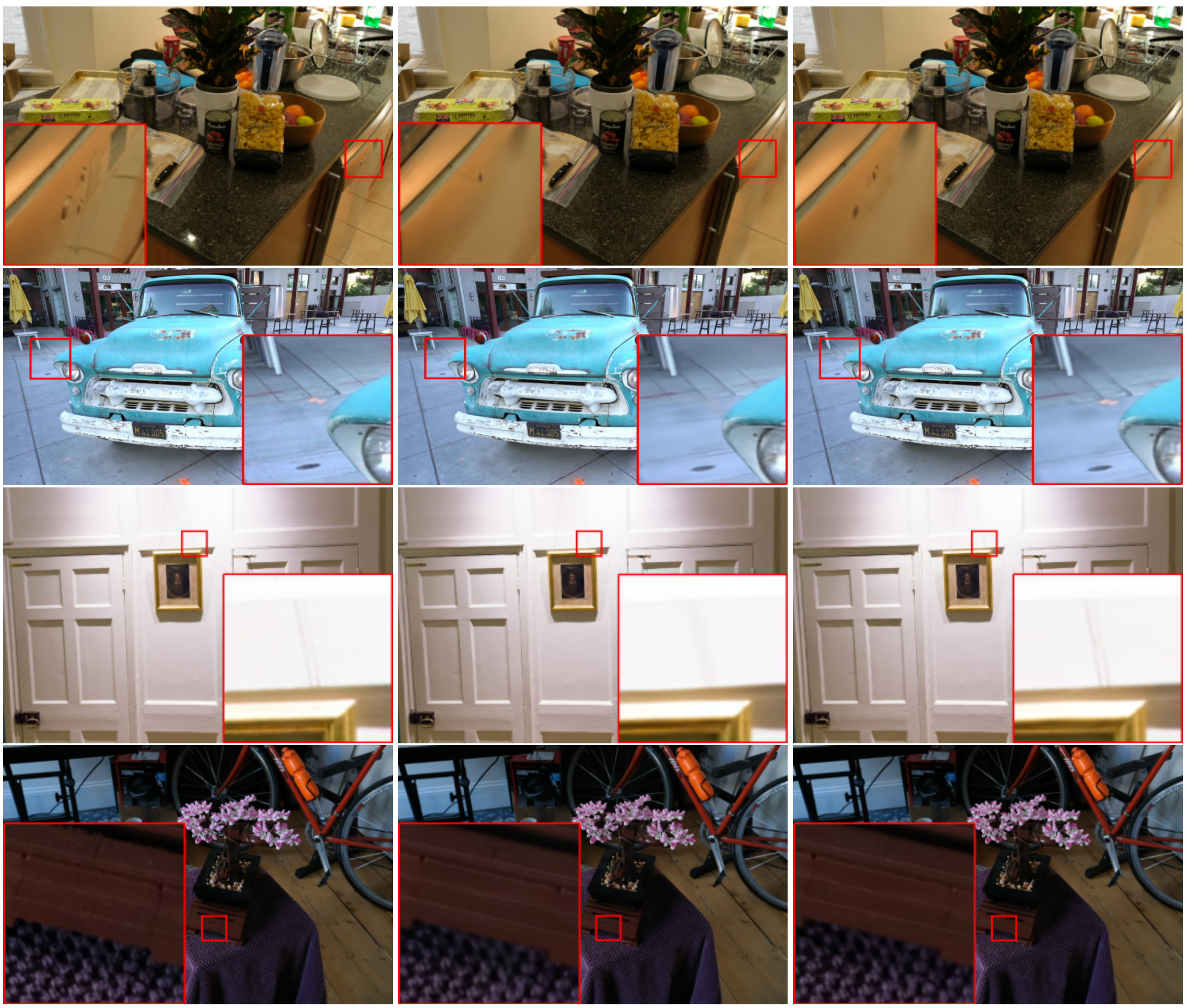

Figure 1. **Qualitative Visualization Across 3DGS and raised cosine splatting.** Displayed are side-by-side comparisons across the Counter, Truck, DrJohnson, Bonsai scenes (top to bottom) repsectively from Mip-NeRF 360, Tanks&Temples and Deep Blending datasets. In the Counter scene, raised cosines outperform Gaussians by better reconstructing the buttons, rendering them more prominently, whereas Gaussians struggle to achieve this even after full training. Similarly, in the Truck scene, raised cosines successfully reconstruct the orange mark on the floor, a detail that Gaussians fail to capture. In the Dr. Johnson scene, our method renders the string of the picture frame with greater clarity, closely resembling the ground truth imagery, while Gaussians fail to achieve the same level of detail. Lastly, in the Bonsai scene, the edges of the pot are more accurately represented by raised cosines as we can see its shadows, producing results that are closer to the ground truth image compared to those achieved with Gaussians. These examples highlight the advantages of raised cosine splatting in capturing finer details in 3D reconstruction compared to Gaussians.

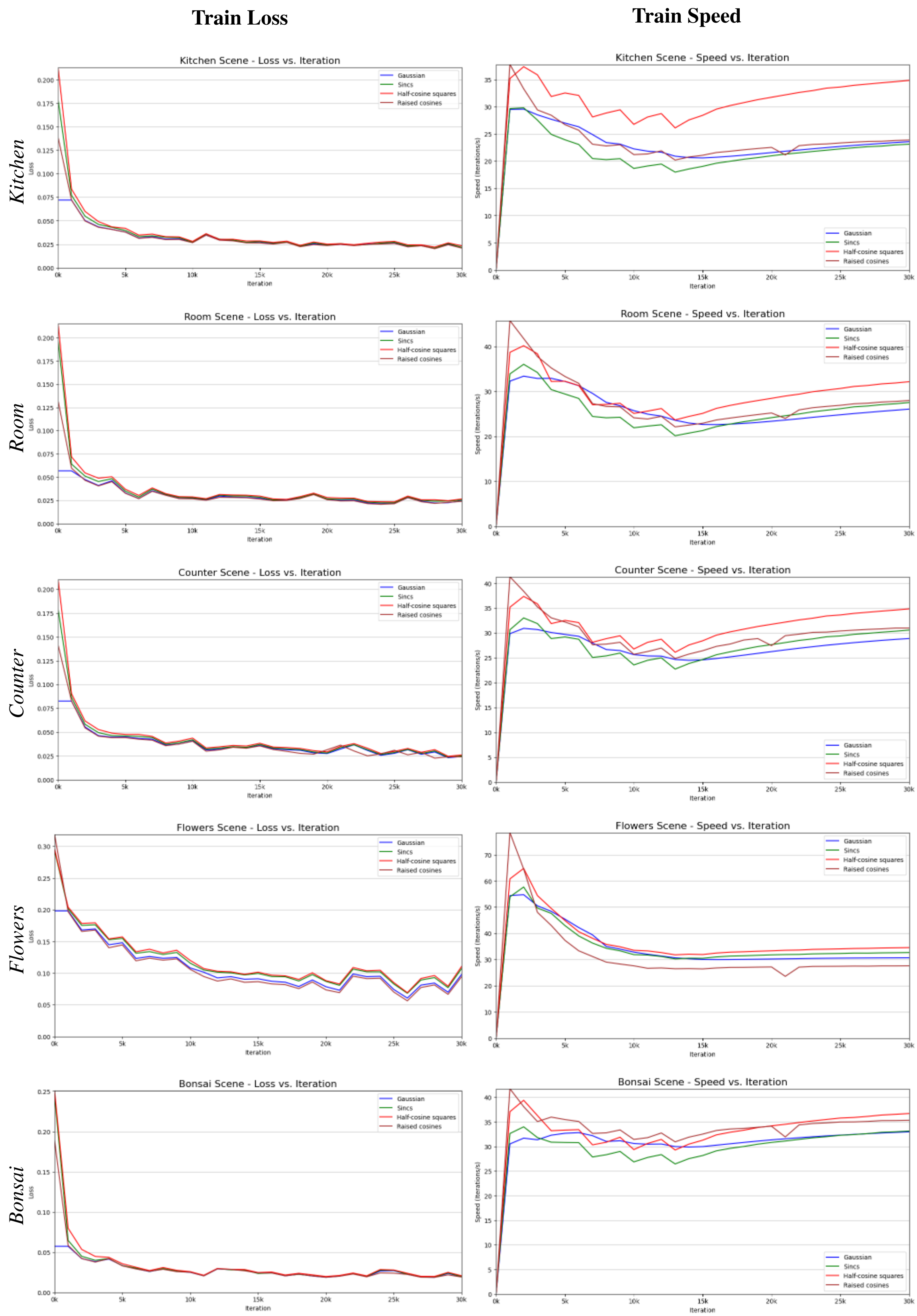

Figure 3. Training loss and speed curves across different scenes reveal significant performance differences. Specifically, the superior convergence speed of half-cosine squared splatting stands out compared to other selected DARBFs, particularly the Gaussian function. Although all the selected functions exhibit similar loss curves, notable variations are observed in their respective training speed curves across various scenes. These differences can be attributed to the inherent characteristics of each scene, which influence the training dynamics of the functions.

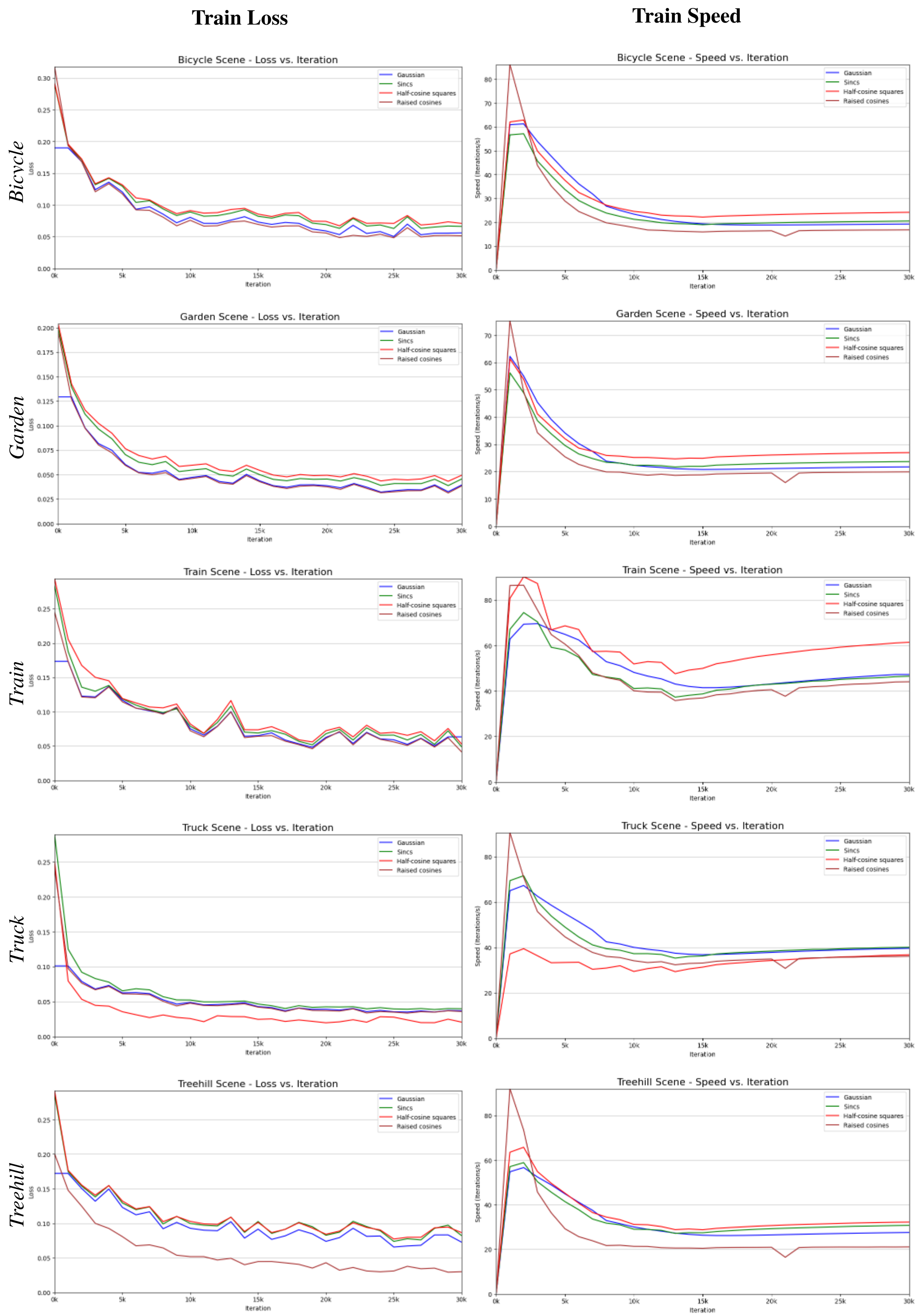

Figure 4. Training loss and speed curves across different scenes reveal significant performance differences. As an exception, in the Truck scene, we observe that the speed curves of Gaussians and raised cosines overlap and outperform half-cosine squares. However, when considering the overall performance across all scenes, half-cosine squares demonstrates superior efficiency in training time.

Table 1. **Performance metrics across various scenes.** Red color denotes best performance, while yellow denotes third-best. Higher values are better for PSNR, SSIM, and lower values are better for LPIPS, Memory, and Training Time. Here we present a detailed breakdown of results for all scenes from Mip-NeRF360 [1] dataset for 7k and 30k iterations. The performance of each model is heavily affected by the nature of the scene. Some values may differ from the main table due to stochastic processes, as these results are from a single instance of a full evaluation experiment series. In contrast, Table **??** presents the mean results averaged across multiple experiments.

Metric	Model	Step	Bicycle	Flowers	Garden	Stump	Treehill	Room	Counter	Kitchen	Bonsai	Mean
PSNR (dB)	3DGS	7k	23.759	20.4657	26.21	25.712	22.09	29.439	27.179	29.213	29.863	25.9525
		30k	25.248	21.519	27.352	26.562	22.554	31.597	29.055	31.378	32.316	27.4509
	3DRCS	7k	23.81	20.52	26.274	25.825	22.09	29.3625	27.261	29.312	29.635	26.0058
		30k	25.222	21.492	27.374	26.542	22.456	31.342	29.032	31.431	31.842	27.4547
	3DHCS	7k	23.037	19.929	25.351	24.821	22.078	29.127	26.837	28.756	29.481	25.4607
		30k	24.369	21.046	26.585	25.937	22.564	31.003	28.782	31.95	31.89	27.0451
	3DSS	7k	23.371	20.177	25.685	25.2	22.134	29.373	27.078	28.88	29.805	25.7447
		30k	24.813	21.2706	26.985	26.262	22.619	31.467	28.982	31.094	32.213	27.3006
	3DIQS	7k	23.309	19.782	25.669	24.893	21.591	28.986	26.71	28.211	28.132	25.2536
		30k	24.885	20.838	26.838	26.169	22.389	31.185	28.547	30.34	30.173	26.8182
LPIPS	3DGS	7k	0.328	0.422	0.16	0.294	0.418	0.26	0.2455	0.157	0.236	0.2801
		30k	0.211	0.342	0.108	0.217	0.33	0.22	0.202	0.126	0.205	0.2178
	3DRCS	7k	0.3197	0.41	0.156	0.283	0.406	0.257	0.241	0.155	0.231	0.2732
		30k	0.2079	0.334	0.108	0.212	0.324	0.219	0.2	0.126	0.203	0.2148
	3DHCS	7k	0.403	0.458	0.233	0.35	0.46	0.2787	0.256	0.168	0.238	0.3161
		30k	0.281	0.368	0.156	0.26	0.375	0.234	0.208	0.132	0.206	0.2466
	3DSS	7k	0.374	0.444	0.206	0.328	0.444	0.271	0.25	0.164	0.237	0.302
		30k	0.249	0.359	0.135	0.241	0.356	0.229	0.205	0.129	0.206	0.2343
	3DIQS	7k	0.355	0.462	0.18	0.335	0.443	0.268	0.253	0.17	0.242	0.3008
		30k	0.238	0.281	0.126	0.248	0.358	0.228	0.21	0.134	0.21	0.2258
SSIM	3DGS	7k	0.669	0.523	0.826	0.722	0.586	0.894	0.875	0.903	0.92	0.7686
		30k	0.763	0.6	0.863	0.769	0.633	0.917	.903	0.9256	0.939	0.8125
	3DRCS	7k	0.6735	0.5302	0.8297	0.7283	0.5913	0.8953	0.8786	0.905	0.9214	0.7726
		30k	0.7641	0.6039	0.864	0.7703	0.6325	0.9176	0.906	0.9256	0.94	0.8137
	3DHCS	7k	0.602	0.478	0.771	0.668	0.557	0.883	0.866	0.894	0.915	0.7371
		30k	0.706	0.566	0.827	0.7337	0.61	0.91	0.898	0.92	0.9355	0.7895
	3DSS	7k	0.631	0.498	0.795	0.693	0.571	0.889	0.873	0.899	0.918	0.7508
		30k	0.736	0.583	0.845	0.752	0.623	0.914	0.903	0.923	0.938	0.8008
	3DIQS	7k	0.637	0.474	0.8	0.678	0.564	0.884	0.862	0.889	0.911	0.7443
		30k	0.739	0.555	0.843	0.742	0.615	0.909	0.894	0.914	0.928	0.7932
Memory (MB)	3DGS	7k	753	503	840	844	502	259	229	358	252	504
		30k	1135	658	950	1024	727	313	250	384	258	633
	3DRCS	7k	789	516	842	879	528	255	238	370	275	521
		30k	1120	668	953	1010	749	343	276	412	279	645
	3DHCS	7k	570	412	630	699	413	211	228	322	211	410
		30k	858	578	739	867	632	256	242	372	245	532
	3DSS	7k	592	424	672	706	396	240	223	364	266	431
		30k	948	586	800	901	602	281	250	393	272	559
	3DIQS	7k	394	263	484	478	245	153	140	223	149	281
		30k	608	373	518	611	375	182	151	238	153	356
Training time (s)	3DGS	7k	181	160	212	173	163	225	240	265	217	204
		30k	1378	920	1302	1149	1020	1178	1107	1313	949	1146
	3DRCS	7k	193	155	218	182	159	194	205	239	192	193
		30k	1581	995	1400	1292	1098	1087	1010	1247	876	1176
	3DHCS	7k	164	145	199	158	145	217	223	252	210	172
		30k	1103	806	1035	996	848	1002	934	1181	870	975
	3DSS	7k	182	153	202	169	157	233	243	281	185	201
		30k	1324	879	1215	1093	944	1200	1106	1413	1001	1130
	3DIQS	7k	161	146	186	152	155	238	263	269	223	199
		30k	1004	704	993	842	799	1105	1049	1261	880	960

Table 2. **Performance metrics across various scenes.** Red color denotes best performance, while yellow denotes third-best. Higher values are better for PSNR, SSIM, and lower values are better for LPIPS, Memory, and Training Time. Here we present a detailed breakdown of results for all scenes from Tanks&Temples [6] and Deep Blending [4] datasets for 7k and 30k iterations. The performance of each model is heavily affected by the nature of the scene. Some values may differ from the main table due to stochastic processes, as these results are from a single instance of a full evaluation experiment series. In contrast, Table ?? presents the mean results averaged across multiple experiments.

Metric	Model	Step	Tanks&Temples			Deep Blending		
			Truck	Train	Mean	DrJohnson	Playroom	Mean
PSNR (dB)	3DGS	7k	23.933	19.795	21.784	27.609	29.354	28.4215
		30k	25.481	22.201	23.771	29.493	29.976	29.6645
	3DRCS	7k	24.026	19.758	21.882	27.437	29.417	28.477
		30k	25.314	22.077	23.6355	29.35	29.981	29.6355
	3DHCS	7k	22.851	19.391	21.071	26.844	28.965	27.9345
		30k	24.561	21.721	23.108	29.003	29.772	29.3875
	3DSS	7k	23.44	19.658	21.589	27.371	29.417	28.394
		30k	25.03	21.973	23.5065	29.414	29.955	29.6645
	3DIQS	7k	23.3	19.43	21.365	27.279	28.904	28.0915
		30k	24.83	21.856	23.343	29.239	29.769	29.504
LPIPS	3DGS	7k	0.197	0.318	0.2515	0.318	0.284	0.301
		30k	0.144	0.199	0.1725	0.237	0.243	0.24
	3DRCS	7k	0.19	0.312	0.251	0.3178	0.282	0.2999
		30k	0.1423	0.196	0.1661	0.238	0.2435	0.2407
	3DHCS	7k	0.235	0.354	0.2945	0.341	0.298	0.3195
		30k	0.169	0.231	0.2	0.25	0.256	0.253
	3DSS	7k	0.218	0.334	0.276	0.325	0.292	0.3085
		30k	0.159	0.218	0.1885	0.243	0.25	0.2465
	3DIQS	7k	0.213	0.34	0.2765	0.327	0.292	0.3095
		30k	0.154	0.221	0.1875	0.24	0.247	0.2435
SSIM	3DGS	7k	0.848	0.719	0.7815	0.87	0.894	0.882
		30k	0.88	0.818	0.851	0.903	0.903	0.903
	3DRCS	7k	0.8527	0.7242	0.7884	0.8696	0.8942	0.8819
		30k	0.8818	0.8197	0.8507	0.9015	0.9013	0.9014
	3DHCS	7k	0.813	0.684	0.7485	0.856	0.887	0.8715
		30k	0.858	0.791	0.8245	0.899	0.9	0.8995
	3DSS	7k	0.831	0.704	0.7675	0.866	0.891	0.8785
		30k	0.869	0.803	0.836	0.902	0.902	0.902
	3DIQS	7k	0.827	0.693	0.76	0.862	0.886	0.874
		30k	0.866	0.795	0.8305	0.902	0.899	0.9005
Memory (MB)	3DGS	7k	406	180	293	462	336	399
		30k	485	257	371	742	412	577
	3DRCS	7k	476	132	304	491	352	421.5
		30k	548	181	364.5	767	446	606.5
	3DHCS	7k	344	145	244.5	355	331	343
		30k	446	230	338	634	417	482.5
	3DSS	7k	355	156	255.5	399	316	357.5
		30k	434	232	333	682	400	541
	3DIQS	7k	236	114	175	299	197	248
		30k	291	156	223.5	489	239	364
Training time (s)	3DGS	7k	132	112	122	212	176	194
		30k	738	631	684.5	1385	1039	1212
	3DRCS	7k	134	105	119.5	193	162	177.5
		30k	793	660	726.5	1378	1023	1200.5
	3DHCS	7k	114	103	108.5	213	169	191
		30k	604	498	551	1163	947	1055
	3DSS	7k	131	118	124.5	219	176	197.5
		30k	745	659	702	1407	1087	1247
	3DIQS	7k	129	123	126	216	164	190
		30k	624	602	613	1262	918	1090

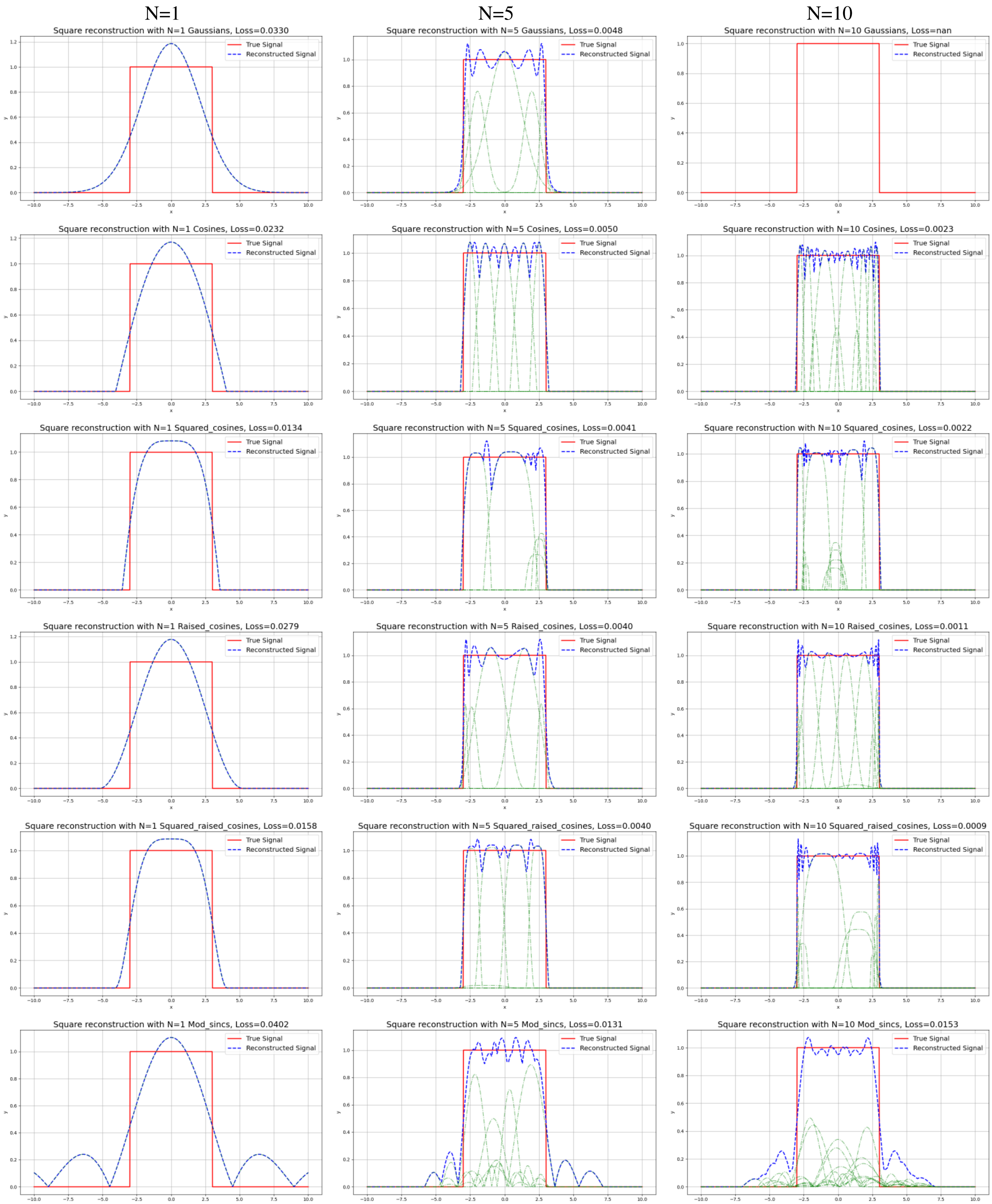

Figure 5. Visualization of 1D simulations for different splatting methods with varying primitives (N=1, N=5, N=10) for a square pulse. Each row corresponds to a specific splatting method: Gaussian, Cosine, Squared Cosine, Raised Cosine, Squared Raised Cosine, and Modulated Sinc. The columns represent the number of primitives used.

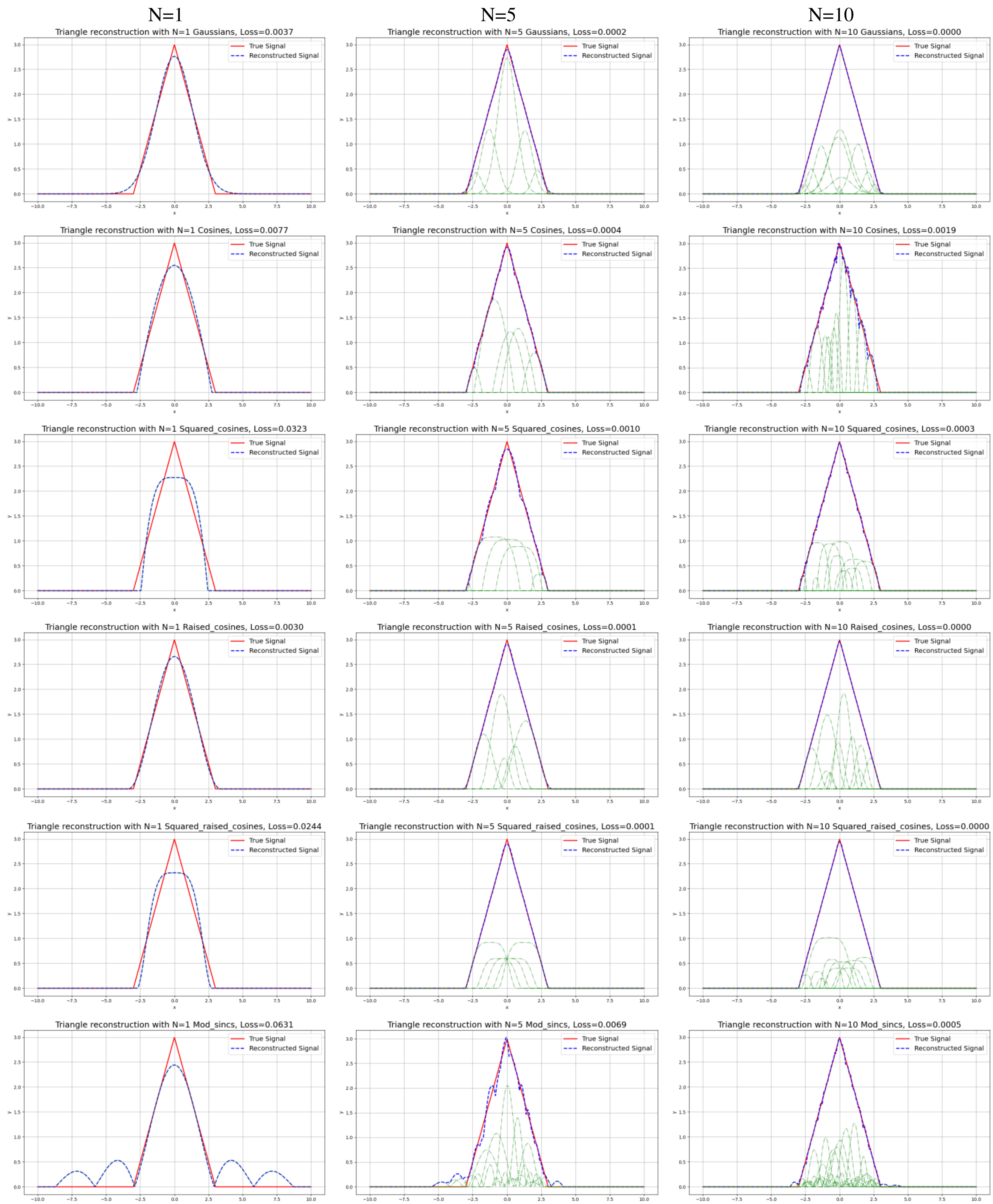

Figure 6. Visualization of 1D simulations for different splatting methods with varying primitives (N=1, N=5, N=10) for a triangle pulse. Each row corresponds to a specific splatting method: Gaussian, Cosine, Squared Cosine, Raised Cosine, Squared Raised Cosine, and Modulated Sinc. The columns represent the number of primitives used.

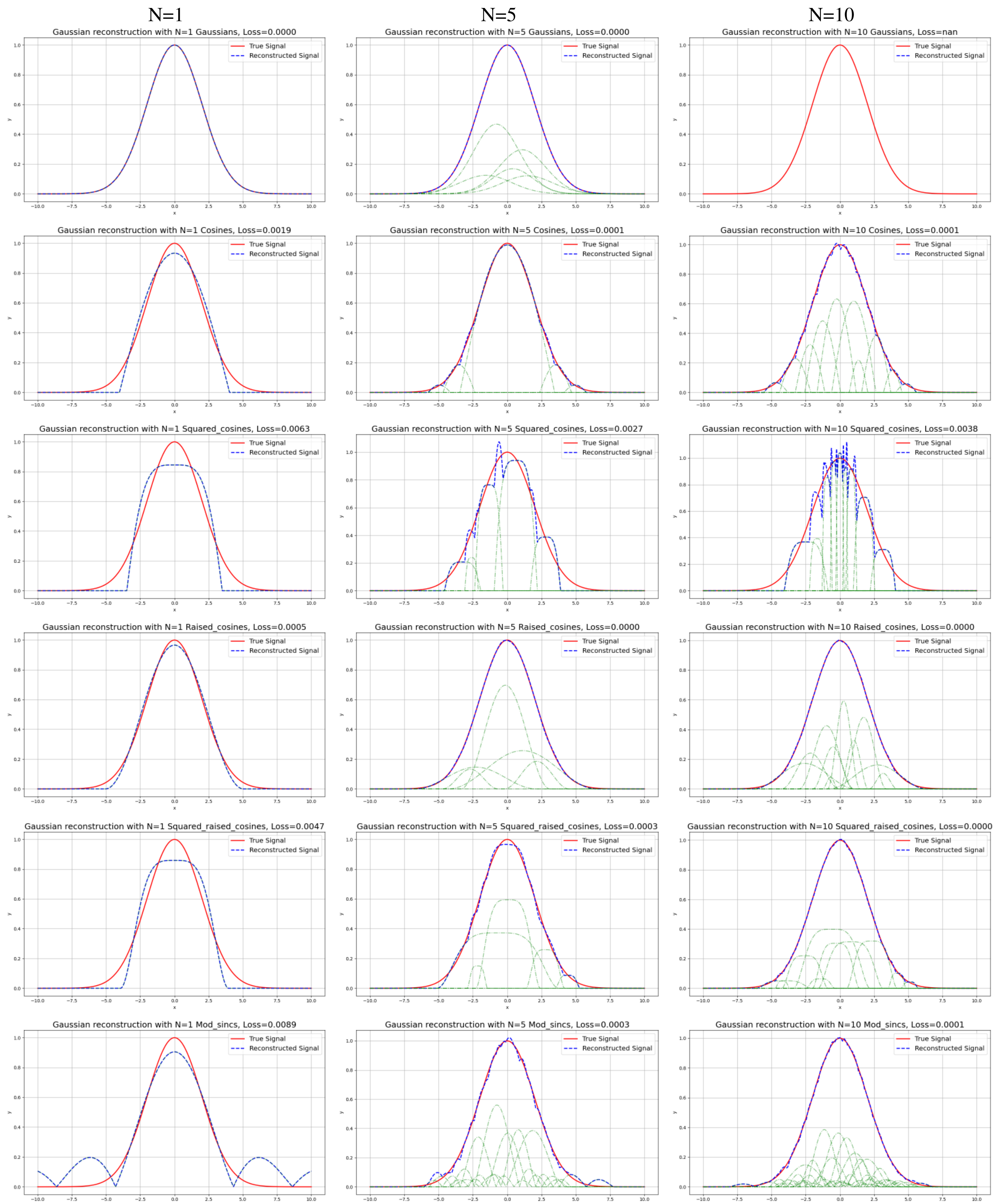

Figure 7. Visualization of 1D simulations for different splatting methods with varying primitives (N=1, N=5, N=10) for a Gaussian. Each row corresponds to a specific splatting method: Gaussian, Cosine, Squared Cosine, Raised Cosine, Squared Raised Cosine, and Modulated Sinc. The columns represent the number of primitives used.

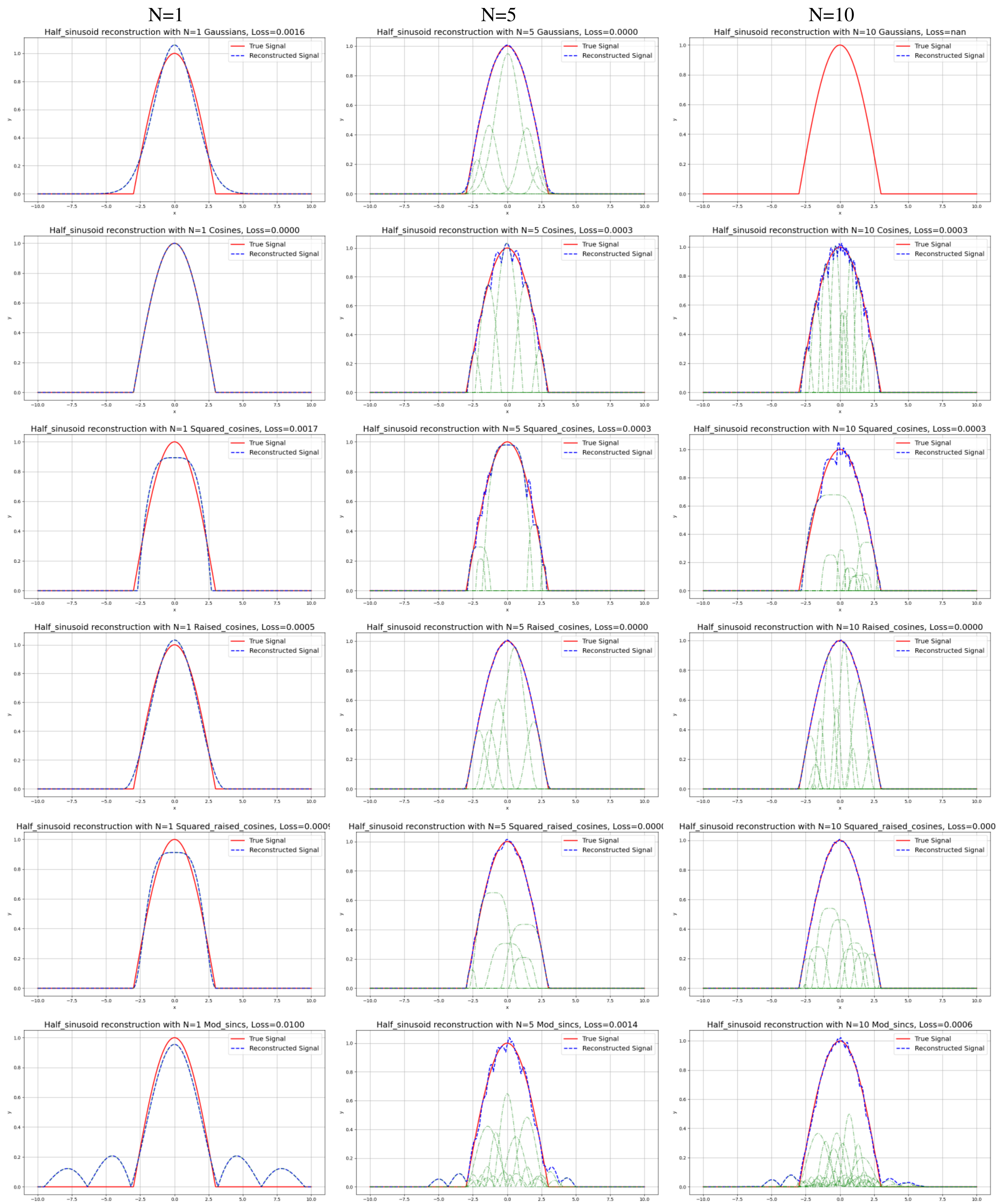

Figure 8. Visualization of 1D simulations for different splatting methods with varying primitives (N=1, N=5, N=10) for a half-sinusoid single pulse. Each row corresponds to a specific splatting method: Gaussian, Cosine, Squared Cosine, Raised Cosine, Squared Raised Cosine, and Modulated Sinc. The columns represent the number of primitives used.

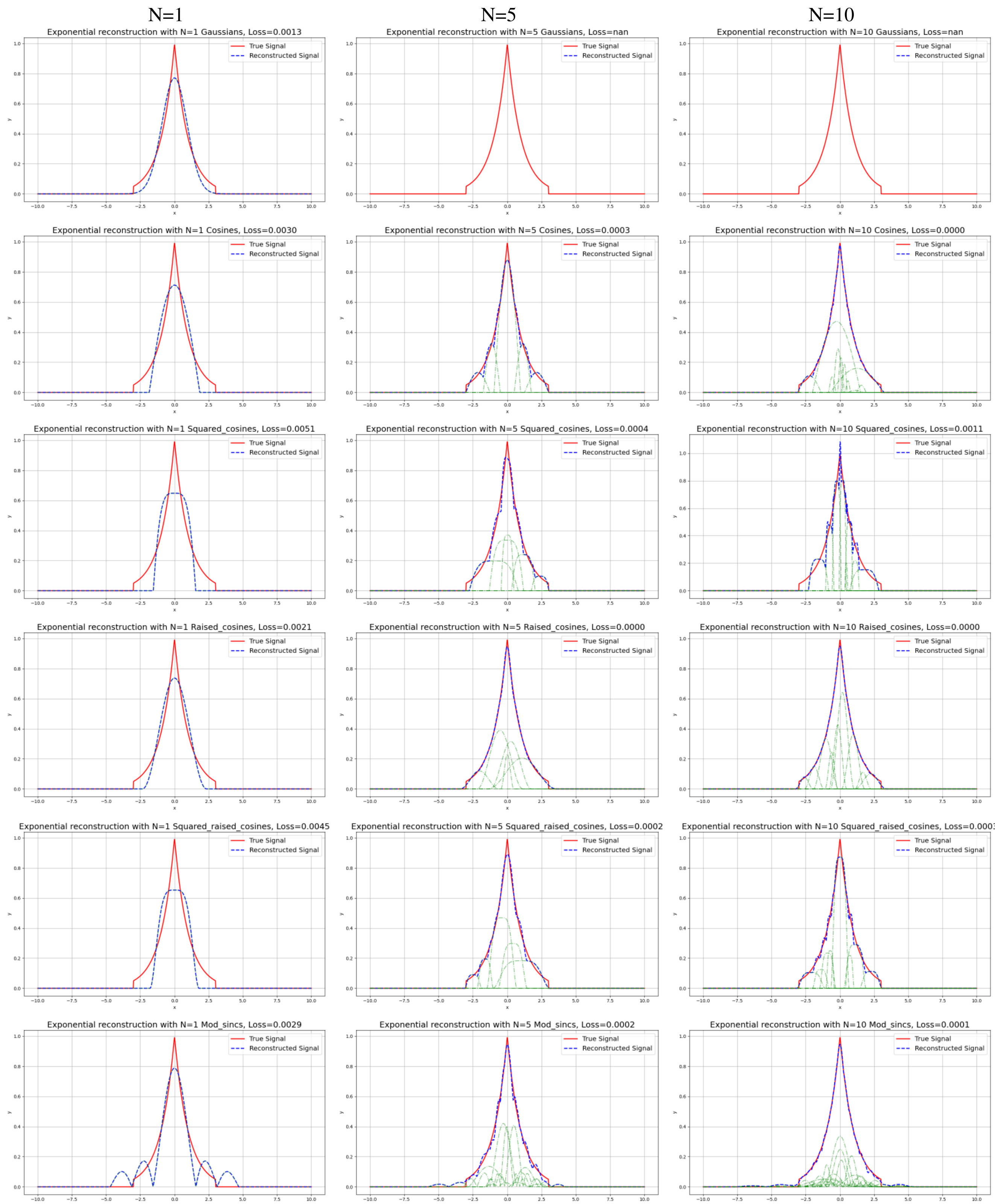

Figure 9. Visualization of 1D simulations for different splatting methods with varying primitives (N=1, N=5, N=10) for a sharp exponential pulse. Each row corresponds to a specific splatting method: Gaussian, Cosine, Squared Cosine, Raised Cosine, Squared Raised Cosine, and Modulated Sinc. The columns represent the number of primitives used.

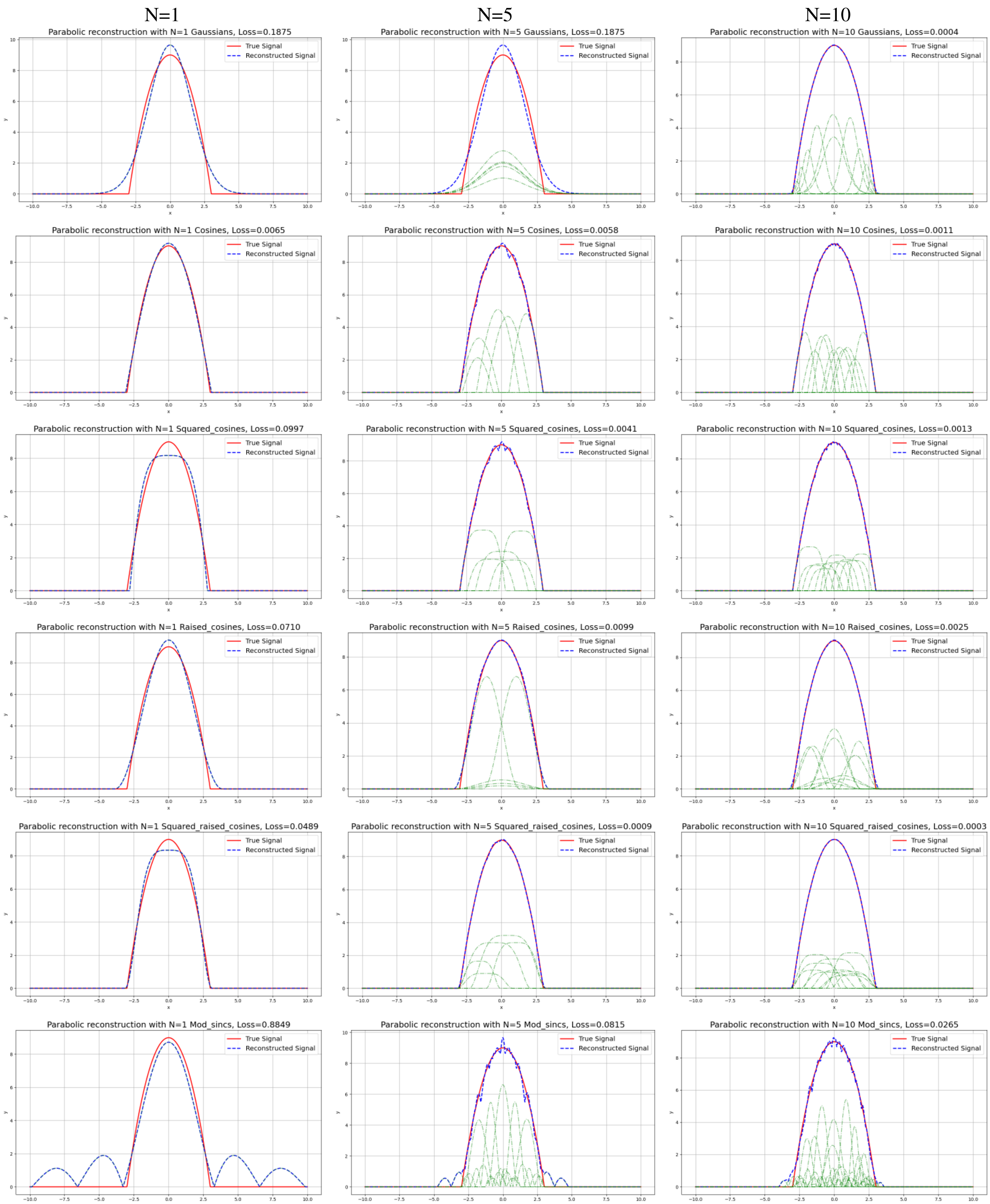

Figure 10. Visualization of 1D simulations for different splatting methods with varying primitives (N=1, N=5, N=10) for a parabolic pulse. Each row corresponds to a specific splatting method: Gaussian, Cosine, Squared Cosine, Raised Cosine, Squared Raised Cosine, and Modulated Sinc. The columns represent the number of primitives used.

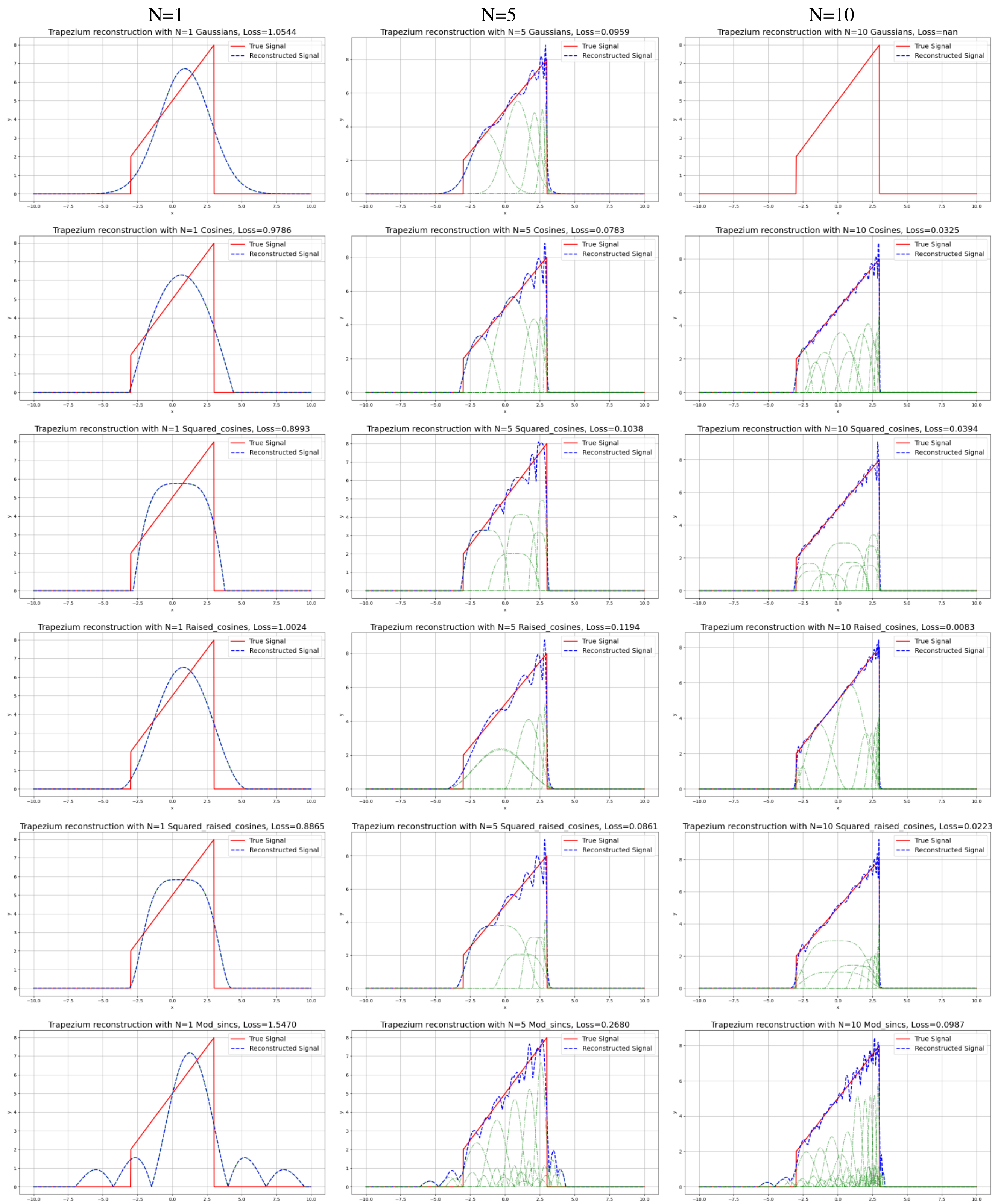

Figure 11. Visualization of 1D simulations for different splatting methods with varying primitives (N=1, N=5, N=10) for a trapezoid pulse. Each row corresponds to a specific splatting method: Gaussian, Cosine, Squared Cosine, Raised Cosine, Squared Raised Cosine, and Modulated Sinc. The columns represent the number of primitives used.