NeRF

A Preprint

1 Main

Algorithm 1 From a particular viewpoint

1: March camera rays through the scene to generate a sampled set of 3D points
2: Use those points and their corresponding 2D viewing directions as input to the neural network to produce an output set of colors and densities
3: Use classical volume rendering techniques to accumulate those colors and densities into a 2D image

*This process is also differentiable, so we can use gradient-descent to optimize model(minimize the error between each observed image and the corresponding views rendered from representation)

"Coherent"refers to the model's ability to represent the 3D scene in a way that is consistent, logical, and unified across different viewpoints.

1.1 Techinal contributions

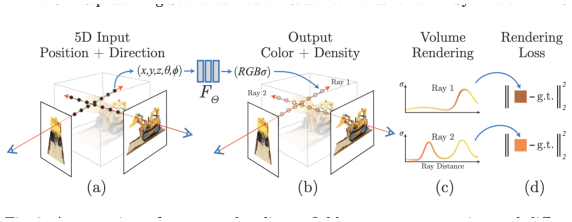
- Representing materials and geometry as 5D parametrized as MLP network
- differentiable rendering procedure based on classical volume rendering techniques, which we use to optimize these representations from standard RGB images. This includes a hierarchical sampling strategy to allocate the MLP's capacity towards space with visible scene content.
- Positional encoding to map 5D into a higher dimensional space

2 Neural Radiance Field Scene Representation

In paper authors suggest 5D representation. It consists of 3D location (x; y; z) and 2D viewing direction $(\theta; \phi)$ and whose output is and emitted color $c = (r; g; b)$ and volume density σ.

Direction is expressed by a 3D Cartesian unit vector d. A vector in a 3D Cartesian coordinate system is defined by three components, typically (dx, dy, dz), corresponding to its projection onto the X, Y, and Z axes, respectively. This vector represents a direction and a magnitude (or length) in 3D space. Unit Vector:

A unit vector is a vector whose magnitude (length) is exactly 1. To convert any non-zero vector into a unit vector in the same direction (a process called normalization), you divide each of its components by its magnitude. The magnitude of a vector (dx, dy, dz) is $\sqrt{(dx^2 + dy^2 + dz^2)}$. So, for a unit vector d = (dx, dy, dz), it holds that $dx^2 + dy^2 + dz^2 = 1$. Using a unit vector is a standard way to represent pure direction, as the length component is fixed at 1 and doesn't carry extra information.



Authors approximate this continuous 5D scene representation with an MLP network $F_\Theta : (\mathbf{x}, \mathbf{d}) \rightarrow (\mathbf{c}, \sigma)$ and optimize its weights Θ to map from each input 5D coordinate to its corresponding volume density and directional emitted color. They encourage the representation to be multiview consistent by restricting the network to predict the volume density as a function of only the location x, while allowing the RGB color c to be predicted as a function of both location and viewing direction.

Algorithm 2 Processing

1: 3D coords x processed with 8 FC layers with ReLU activation and dim(features) = 256 per layer. Output: σ and 256-dim feature vector.
2: Feature vector is concatinated with the camera ray's viewing direction and passed to FC layer (dim(features) = 128 per channel and ReLU)
3: Use classical volume rendering techniques to accumulate those colors and densities into a 2D image. Output: view-dependent RGB color

3 Volume Rendering with Radiance Fields

Vocabulary:

- Volume density $\sigma(x)$(differential probability of a ray terminating at an infinitesimal particle at location x)
- $$C(r) \triangleq \int_{t_n}^{t_f} T(t)\sigma(r(t))c(r(t); d)dt$$
  - expected color, where $T(t) \triangleq \exp(-\int_{t_n}^{t} \sigma(r(s))ds)$
- $r(t) \triangleq o + td$ (camera ray) with near and far bounds t_n and t_f

What is meant by expected color formula is:

- $C(r)$ is the final color observed by the camera along the ray r.
- $r(t) = o + td$ is a point in 3D space at distance t along the ray, where o is the ray origin and d is its direction.
- t_n and t_f are the near and far bounds of the ray integration.
- $\sigma(r(t))$ is the volume density at point $r(t)$ — it represents the probability per unit distance that the ray intersects matter at that point.
- $c(r(t), d)$ is the RGB radiance emitted at point $r(t)$ in direction d. This accounts for view-dependent effects.
- $T(t)$ is the accumulated transmittance from t_n to t, i.e., the probability that the ray travels from t_n to t without hitting anything:
$$T(t) = \exp\left(-\int_{t_n}^{t} \sigma(r(s))\, ds\right) \tag{1}$$
This term decreases as the accumulated density increases, modeling occlusion.

The integrand $\sigma(r(t))c(r(t), d)$ gives the amount of radiance emitted at each point weighted by its density, and $T(t)$ ensures only unoccluded contributions reach the camera.

They also suggest stratified sampling approach for partion $[t_n, t_f]$ into N evenly-spaced bins and then draw one sample uniformlyat random from within each bin:

$$t_i \sim \mathcal{U}[t_n + \frac{i-1}{N}(t_f - t_n), t_n + \frac{i}{N}(t_f - t_n)]$$

By evaluating the MLP at continuously varying positions during optimization, stratified sampling facilitates the representation of the scene as a continuous function.

It is also possible to estimate $C(r)$ as:

$$\hat{C}(r) = \sum_{i=1}^{N} T_i(1 - \exp(-\sigma_i\delta_i))c_i, \text{ where } T_i = \exp(-\sum_{j=1}^{i-1}\sigma_j\delta_j)$$

δ_i is the distance between adjacent samples and stated by equation $\delta_i \triangleq t_{i+1} - t_i$. Previously stated function $\hat{C}(r)$ represents calculating from set (c_i, σ_i) values, it is also differentiable and reduces to traditional alpha compositing with alpha value $\alpha \triangleq 1 - \exp(-\sigma_i\delta_i)$

3.1 Hierarchical Volume Sampling

The naive strategy of densely evaluating the neural radiance field at N_c points along each camera ray is computationally inefficient. This approach allocates equal effort to regions of empty space and occluded areas that contribute little or nothing to the final image. To improve efficiency, the authors draw inspiration from classical volume rendering techniques and propose a hierarchical sampling strategy that concentrates samples in regions with higher expected visual contribution.

Instead of relying on a single network to represent the scene, two networks are trained jointly: a coarse network and a fine network. The coarse network first samples N_c points along each ray using stratified sampling and evaluates the radiance field at these locations (as described in Equations 2 and 3 of the original paper). The resulting output is then used to guide a second sampling step, which focuses on parts of the ray that are more likely to contain visible content.

To achieve this, the alpha-composited color from the coarse network, $\hat{C}_c(\mathbf{r})$, is re-expressed as a weighted sum of the predicted colors $\mathbf{c}_i$ at each sampled point along the ray:

$$\hat{C}_c(\mathbf{r}) = \sum_{i=1}^{N_c} w_i\mathbf{c}_i, \quad \text{where} \quad w_i = T_i\left(1 - \exp(-\sigma_i\delta_i)\right). \tag{2}$$

By normalizing the weights:

$$\hat{w}_i = \frac{w_i}{\sum_{j=1}^{N_c} w_j},$$

a piecewise-constant probability density function (PDF) is formed along the ray. A second set of N_f samples is then drawn from this distribution using inverse transform sampling. These new points, together with the original N_c coarse points, are used to evaluate the fine network, which produces a refined color estimate $\hat{C}_f(\mathbf{r})$ using the same volume rendering procedure.

This strategy is conceptually similar to importance sampling, but instead of treating each sample as an independent Monte Carlo estimate of the full integral, the method uses the weights to non-uniformly discretize the integration domain.

3.2 Implementation Details

A separate neural network is trained for each scene to model its continuous volumetric representation. The training process requires only a set of RGB images, their corresponding camera poses and intrinsics, and the bounding volume of the scene. For synthetic scenes, ground-truth poses and bounds are available, whereas for real-world scenes, these parameters are estimated using the COLMAP structure-from-motion system.

At each optimization step, a random batch of 4096 camera rays is sampled from across all training images. For each ray, $N_c = 64$ points are sampled and passed through the coarse network, and an additional $N_f = 128$ samples are drawn according to the hierarchical PDF and processed by the fine network.

The final rendered color for each ray is computed using both sets of outputs, and the loss function is defined as the sum of squared errors between predicted and ground truth pixel colors for both coarse and fine outputs:

$$\mathcal{L} = \sum_{\mathbf{r}\in\mathcal{R}} \left[\left\| \hat{C}_c(\mathbf{r}) - C(\mathbf{r}) \right\|_2^2 + \left\| \hat{C}_f(\mathbf{r}) - C(\mathbf{r}) \right\|_2^2 \right], \tag{3}$$

where $\mathcal{R}$ denotes the batch of rays, $C(\mathbf{r})$ is the ground-truth color, and $\hat{C}_c(\mathbf{r})$, $\hat{C}_f(\mathbf{r})$ are the outputs from the coarse and fine networks, respectively.

Although only $\hat{C}_f(\mathbf{r})$ is used for final rendering, the coarse loss is also minimized to ensure that its weights guide effective resampling.

The model is optimized using the Adam optimizer with an initial learning rate of 5×10^{-4}, which exponentially decays to 5×10^{-5} over training. The default values of $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-7}$ are used. Training a single scene typically requires 100k–300k iterations, taking approximately 1–2 days on a single NVIDIA V100 GPU.

3.3 Positional Encoding for High-Frequency Detail

Although neural networks are theoretically universal function approximators, we observe in practice that directly feeding raw input coordinates (x, y, z, θ, ϕ) into the MLP F_Θ results in suboptimal performance when modeling high-frequency variations in color and geometry. This observation is in line with the findings of Rahaman et al. [?], who demonstrate that deep networks exhibit a spectral bias, favoring the learning of low-frequency functions over high-frequency ones.

To address this limitation, they propose applying a transformation that maps input coordinates through a set of high-frequency basis functions prior to feeding them into the network. Following this idea, the NeRF architecture adopts a similar strategy within the context of neural scene representations. Specifically, the network F_Θ is reformulated as the composition of two functions:

$$F_\Theta = F'_\Theta \circ \gamma, \tag{4}$$

where γ is a fixed (non-learned) mapping from $\mathbb{R}$ to a higher-dimensional space $\mathbb{R}^{2L}$, and F'_Θ is a standard MLP. This positional encoding γ improves the network's ability to represent fine detail and high-frequency variation in the scene.

The encoding function $\gamma(p)$ is defined as:

$$\gamma(p) = \big(\sin(2^0\pi p), \cos(2^0\pi p), \sin(2^1\pi p), \cos(2^1\pi p), \ldots, \sin(2^{L-1}\pi p), \cos(2^{L-1}\pi p)\big), \tag{5}$$

and is applied independently to each component of the input position vector $\mathbf{x} \in [-1, 1]^3$, as well as the viewing direction unit vector (θ, ϕ).

This encoding allows the MLP to better capture high-frequency scene content such as sharp edges, fine textures, and view-dependent reflectance effects, which would otherwise be underfit due to the spectral limitations of standard architectures.

4 NDC ray space derivation

To reconstruct real-world forward-facing scenes, NeRF employs the normalized device coordinate (NDC) space—a canonical space often used in the graphics pipeline for triangle rasterization. NDC space offers two major benefits: it maintains parallelism in lines and transforms the depth axis into a linear representation of disparity (inverse depth), which is more suitable for learning from perspective imagery.

Here, we outline the transformation that maps rays from camera coordinates into NDC space. The mapping is derived from the standard perspective projection matrix applied to homogeneous 3D coordinates. The 4×4 projection matrix $\mathbf{M}$ is:

$$\mathbf{M} = \begin{pmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}, \tag{6}$$

where n and f are the near and far clipping planes, and r and t define the right and top bounds of the frustum at the near plane. We assume the camera looks along the $-z$ direction.

To project a point $\mathbf{p} = (x, y, z, 1)^T$ into NDC space, we left-multiply by $\mathbf{M}$ and divide by the fourth coordinate:

$$\mathbf{M} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{n}{r}x \\ \frac{n}{t}y \\ -\frac{f+n}{f-n}z - \frac{2fn}{f-n} \\ -z \end{pmatrix} \Rightarrow \begin{pmatrix} \frac{n}{r}\frac{x}{-z} \\ \frac{n}{t}\frac{y}{-z} \\ \frac{f+n}{f-n} + \frac{2fn}{(f-n)z} \\ 1 \end{pmatrix}. \tag{7}$$

This maps the original viewing frustum to the unit cube $[-1, 1]^3$.

Our goal is to transform a ray of the form $\mathbf{o} + t\mathbf{d}$ in camera space into a corresponding ray $\mathbf{o}' + t'\mathbf{d}'$ in NDC space such that both rays trace out the same set of points under projection $\pi(\cdot)$.

To simplify, we define the projected point as:

$$\pi(x, y, z) = \left(\frac{a_x x}{z}, \frac{a_y y}{z}, \frac{a_z z + b_z}{z} \right),$$

and set the ray origin in NDC space to be the projection of the original origin:

$$\mathbf{o}' = \begin{pmatrix} \frac{a_x o_x}{o_z} \\ \frac{a_y o_y}{o_z} \\ \frac{a_z o_z + b_z}{o_z} \end{pmatrix} = \pi(\mathbf{o}). \tag{8}$$

To compute the direction $\mathbf{d}'$, we consider an arbitrary point $\mathbf{o} + t\mathbf{d}$ and subtract the origin projection:

$$t'\mathbf{d}' = \begin{pmatrix} \frac{a_x(o_x + td_x)}{o_z + td_z} - \frac{a_x o_x}{o_z} \\ \frac{a_y(o_y + td_y)}{o_z + td_z} - \frac{a_y o_y}{o_z} \\ \frac{a_z(o_z + td_z) + b_z}{o_z + td_z} - \frac{a_z o_z + b_z}{o_z} \end{pmatrix}. \tag{9}$$

After algebraic simplification, this becomes:

$$\mathbf{d}' = \begin{pmatrix} a_x \left(\frac{d_x}{d_z} - \frac{o_x}{o_z} \right) \\ a_y \left(\frac{d_y}{d_z} - \frac{o_y}{o_z} \right) \\ -\frac{b_z}{o_z} \end{pmatrix}, \quad t' = \frac{td_z}{o_z + td_z} = 1 - \frac{o_z}{o_z + td_z}. \tag{10}$$

This ensures that $t' = 0$ when $t = 0$, and $t' \to 1$ as $t \to \infty$. Thus, sampling uniformly in t' corresponds to sampling linearly in disparity in the original space.

From the projection matrix, we identify the coefficients:

$$a_x = -\frac{n}{r}, \qquad a_y = -\frac{n}{t}, \tag{11}$$

$$a_z = \frac{f+n}{f-n}, \qquad b_z = \frac{2fn}{f-n}. \tag{12}$$

Under a pinhole camera model with image dimensions $W \times H$ and focal length f_{cam}, we can reparameterize:

$$a_x = -\frac{f_{\text{cam}}}{W/2}, \quad a_y = -\frac{f_{\text{cam}}}{H/2}. \tag{13}$$

In the forward-facing setup used in NeRF, we set $f \to \infty$, which simplifies the depth-related constants:

$$a_z = 1, \quad b_z = 2n. \tag{14}$$

Putting everything together, the ray transformation to NDC space becomes:

$$\mathbf{o}' = \begin{pmatrix} -\frac{f_{\text{cam}}}{W/2} \cdot \frac{o_x}{o_z} \\ -\frac{f_{\text{cam}}}{H/2} \cdot \frac{o_y}{o_z} \\ 1 + \frac{2n}{o_z} \end{pmatrix}, \quad \mathbf{d}' = \begin{pmatrix} -\frac{f_{\text{cam}}}{W/2} \left(\frac{d_x}{d_z} - \frac{o_x}{o_z} \right) \\ -\frac{f_{\text{cam}}}{H/2} \left(\frac{d_y}{d_z} - \frac{o_y}{o_z} \right) \\ -\frac{2n}{o_z} \end{pmatrix}. \tag{15}$$

Implementation detail. Before applying this transformation, NeRF first shifts the ray origin to intersect the near plane $z = -n$. This is achieved by computing a new origin $\mathbf{o}_n = \mathbf{o} + t_n \mathbf{d}$, where:

$$t_n = -\frac{n + o_z}{d_z}.$$

After this shift, uniform sampling in $t' \in [0, 1]$ corresponds to linear sampling in disparity in the original 3D scene.

Algorithm 3 NeRF Training and Rendering

Inputs: Training images $\{I_k\}$, camera poses $\{P_k\}$, intrinsics (for NDC), near/far bounds t_{near}, t_{far}.
Initialize: Coarse network F_{Θ_c}, Fine network F_{Θ_f}.
procedure RenderColorForRay($\mathbf{o}, \mathbf{d}, F_{\Theta_c}, F_{\Theta_f}, t_{near}, t_{far}$)
    if using NDC then
        $(\mathbf{o}, \mathbf{d}) \leftarrow$ TransformToNDC$(\mathbf{o}, \mathbf{d})$
        $(t_{near}, t_{far}) \leftarrow (0, 1)$
    end if
▷ Coarse Pass
    Sample N_c points $\{t_{c,i}\}_{i=1}^{N_c}$ along ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ from t_{near} to t_{far} using stratified sampling.
    $\forall i \in [1, N_c]$: $\mathbf{x}_{c,i} = \mathbf{o} + t_{c,i}\mathbf{d}$. Query $(\mathbf{c}_{c,i}, \sigma_{c,i}) = F_{\Theta_c}(\mathbf{x}_{c,i}, \mathbf{d})$.
    Compute rendered color $\hat{C}_c(\mathbf{r})$ and weights $w_{c,i}$ from $(\mathbf{c}_{c,i}, \sigma_{c,i})$:
    $\delta_{c,i} = t_{c,i+1} - t_{c,i}$ (distance between adjacent samples, $t_{c,N_c+1} = t_{far}$).
    $\alpha_{c,i} = 1 - \exp(-\sigma_{c,i}\delta_{c,i})$.
    $T_{c,i} = \exp\left(-\sum_{j=1}^{i-1} \sigma_{c,j}\delta_{c,j}\right) = \prod_{j=1}^{i-1}(1 - \alpha_{c,j})$. $(T_{c,1} = 1)$.
    $\hat{C}_c(\mathbf{r}) = \sum_{i=1}^{N_c} T_{c,i}\alpha_{c,i}\mathbf{c}_{c,i}$.
    $w_{c,i} = T_{c,i}\alpha_{c,i}$.
▷ Fine Pass
    Normalize weights: $\hat{w}_{c,i} = w_{c,i} / \sum_{j=1}^{N_c} w_{c,j}$. This forms a PDF over the coarse bins.
    Sample N_f points $\{t_{f_i}\}_{i=1}^{N_f}$ from this PDF using inverse transform sampling.
    Combine and sort all samples: $\{t_i\}_{i=1}^{N_c+N_f} = \text{Sort}(\{t_{c,i}\} \cup \{t_{f,i}\})$.
    $\forall i \in [1, N_c + N_f]$: $\mathbf{x}_i = \mathbf{o} + t_i\mathbf{d}$. Query $(\mathbf{c}_i, \sigma_i) = F_{\Theta_f}(\mathbf{x}_i, \mathbf{d})$.
    Compute rendered color $\hat{C}_f(\mathbf{r})$ from $(\mathbf{c}_i, \sigma_i)$:
    $\delta_i = t_{i+1} - t_i$.
    $\alpha_i = 1 - \exp(-\sigma_i\delta_i)$.
    $T_i = \exp\left(-\sum_{j=1}^{i-1} \sigma_j\delta_j\right) = \prod_{j=1}^{i-1}(1 - \alpha_j)$.
    $\hat{C}_f(\mathbf{r}) = \sum_{i=1}^{N_c+N_f} T_i\alpha_i\mathbf{c}_i$.
    return $(\hat{C}_c(\mathbf{r}), \hat{C}_f(\mathbf{r}))$.
end procedure

procedure TrainNeRF
    for each training iteration do
        Sample a batch of rays $\mathcal{R}$ from pixels of $\{I_k\}$, with origins $\mathbf{o}_k$ and directions $\mathbf{d}_k$.
        $\mathcal{L} = 0$.
        for each ray $(\mathbf{o}, \mathbf{d})$ with ground truth color $C(\mathbf{r})$ in $\mathcal{R}$ do
            $(\hat{C}_c(\mathbf{r}), \hat{C}_f(\mathbf{r})) =$ RenderColorForRay$(\mathbf{o}, \mathbf{d}, F_{\Theta_c}, F_{\Theta_f}, t_{near}, t_{far})$.
            $\mathcal{L} \mathrel{+}= ||\hat{C}_c(\mathbf{r}) - C(\mathbf{r})||_2^2 + ||\hat{C}_f(\mathbf{r}) - C(\mathbf{r})||_2^2$.
        end for
        Update Θ_c, Θ_f by taking a gradient step on $\mathcal{L}$ (e.g., using Adam optimizer).
    end for
end procedure

procedure RenderNovelView(Target camera pose P_{target})
    Initialize empty image I_{novel}.
    for each pixel (u, v) in I_{novel} do
        Compute ray $(\mathbf{o}, \mathbf{d})$ from P_{target} passing through (u, v).
        $(\_, \hat{C}_f(\mathbf{r})) =$ RenderColorForRay$(\mathbf{o}, \mathbf{d}, F_{\Theta_c}, F_{\Theta_f}, t_{near}, t_{far})$.
        $I_{novel}[u, v] = \hat{C}_f(\mathbf{r})$.
    end for
    return I_{novel}.
end procedure

Algorithm 4 NeRF Inference: Render Novel View

function RenderRayForInference($\mathbf{o}, \mathbf{d}, F_{\Theta_c}, F_{\Theta_f}, t_{near}, t_{far}, N_c, N_f$, use_ndc)
  $\mathbf{o}_{render} \leftarrow \mathbf{o}$, $\mathbf{d}_{render} \leftarrow \mathbf{d}$
  $t_{n\_render} \leftarrow t_{near}$, $t_{f\_render} \leftarrow t_{far}$
  if use_ndc is True then
    $(\mathbf{o}_{render}, \mathbf{d}_{render}) \leftarrow$ TransformToNDC($\mathbf{o}, \mathbf{d}$, near_plane_dist, focal, img_W, img_H)
    $t_{n\_render} \leftarrow 0$, $t_{f\_render} \leftarrow 1$
  end if
  ▷ Coarse Pass
  Sample N_c points $\{t_{c,i}\}_{i=1}^{N_c}$ along ray $\mathbf{r}(t) = \mathbf{o}_{render} + t\mathbf{d}_{render}$ from t_{n_render} to t_{f_render} using stratified sampling.
  Initialize lists: $C_c = []$, $\Sigma_c = []$.
  for $i = 1$ to N_c do
    $\mathbf{x}_{c,i} = \mathbf{o}_{render} + t_{c,i}\mathbf{d}_{render}$
    $(\mathbf{c}_{c,i}, \sigma_{c,i}) = F_{\Theta_c}(\gamma(\mathbf{x}_{c,i}), \gamma(\mathbf{d}_{render}))$
    Append $\mathbf{c}_{c,i}$ to C_c, $\sigma_{c,i}$ to Σ_c.
  end for
  Compute weights $w_{c,i}$ from $(C_c, \Sigma_c, \{t_{c,i}\})$:
  for $i = 1$ to N_c do
    $\delta_{c,i} = (i < N_c)?(t_{c,i+1} - t_{c,i}) : (t_{f\_render} - t_{c,N_c})$
    $\alpha_{c,i} = 1 - \exp(-\sigma_{c,i}\delta_{c,i})$
    $T_{c,i} = \prod_{j=1}^{i-1}(1 - \alpha_{c,j})$ ▷ $T_{c,1} = 1$
    $w_{c,i} = T_{c,i}\alpha_{c,i}$
  end for
  ▷ Fine Pass
  Normalize weights: $\hat{w}_{c,i} = w_{c,i} / \sum_{j=1}^{N_c} w_{c,j}$. This forms a PDF over the coarse sample intervals.
  Sample N_f points $\{t_{f,j}\}_{j=1}^{N_f}$ from this PDF using inverse transform sampling (sampling from the midpoints of the coarse intervals weighted by $\hat{w}_{c,i}$).
  Combine and sort all samples: $\{t_k\}_{k=1}^{N_c+N_f} = \text{Sort}(\{t_{c,i}\} \cup \{t_{f,j}\})$.
  Initialize lists: $C_{fine} = []$, $\Sigma_{fine} = []$.
  for $k = 1$ to $N_c + N_f$ do
    $\mathbf{x}_k = \mathbf{o}_{render} + t_k\mathbf{d}_{render}$
    $(\mathbf{c}_k, \sigma_k) = F_{\Theta_f}(\gamma(\mathbf{x}_k), \gamma(\mathbf{d}_{render}))$
    Append $\mathbf{c}_k$ to C_{fine}, σ_k to Σ_{fine}.
  end for
  Compute final rendered color $\hat{C}_{final}(\mathbf{r})$ from $(C_{fine}, \Sigma_{fine}, \{t_k\})$:
  $\hat{C}_{final}(\mathbf{r}) \leftarrow \mathbf{0}$ ▷ Initialize color (e.g., black)
  for $k = 1$ to $N_c + N_f$ do
    $\delta_k = (k < N_c + N_f)?(t_{k+1} - t_k) : (t_{f\_render} - t_{N_c+N_f})$ ▷ Adjust last delta if needed
    if $\delta_k = 0$ and $k = N_c + N_f$ then ▷ Handle edge case of last sample
      $\delta_k = 1e - 5$ ▷ Small epsilon if last sample is at t_{far}
    end if
    $\alpha_k = 1 - \exp(-\sigma_k\delta_k)$
    $T_k = \prod_{j=1}^{k-1}(1 - \alpha_j)$ ▷ $T_1 = 1$
    $\hat{C}_{final}(\mathbf{r}) \mathrel{+}= T_k\alpha_k\mathbf{c}_k$
  end for
  return $\hat{C}_{final}(\mathbf{r})$
end function

procedure GenerateNovelView(P_{target}, Intrinsics, $W_{img}, H_{img}, F_{\Theta_c}, F_{\Theta_f}$, params)
  Initialize empty image I_{novel} of size $W_{img} \times H_{img}$.
  Extract $t_{near}, t_{far}, N_c, N_f$, use_ndc, etc. from params.
  for $y = 0$ to $H_{img} - 1$ do
    for $x = 0$ to $W_{img} - 1$ do
      Compute ray origin $\mathbf{o}$ and direction $\mathbf{d}$ for pixel (x, y) using P_{target} and Intrinsics.
      $PixelColor \leftarrow$ RenderRayForInference($\mathbf{o}, \mathbf{d}, F_{\Theta_c}, F_{\Theta_f}, t_{near}, t_{far}, N_c, N_f$, use_ndc).
      $I_{novel}[x, y] \leftarrow PixelColor$.
    end for
  end for
  return I_{novel}.
end procedure