Neural 3D Strokes: Creating Stylized 3D Scenes with Vectorized 3D Strokes

Supplementary Material

This supplementary document provides additional details of the 3D strokes in Sec. 6, implementations in Sec.7, additional comparisons of various 3D strokes in Sec. 8, and application training setups in Sec. 9. Please also watch our accompanying video for an animated visualization of stylization results.

6. Details of 3D Strokes

6.1. Transformation of Basic Primitives

In Sec. 3.2.1 of the main paper, we use a transformation matrix to map the coordinates in the shared scene space into the canonical space of each unit signed distance field. Here we provide the construction details of the transformation matrix. Given a translation vector $\mathbf{t} = (t_x, t_y, t_z)$, an Euler angle rotation vector $\mathbf{r} = (r_x, r_y, r_z)$, and a scale vector $\mathbf{s} = (s_x, s_y, s_z)$, we first construct the matrices for each term respectively, then combine them in the order of scale, rotation, and translation to get the final transformation matrix M:

$$
\begin{aligned}
T &= \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}, \\
R_x &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos r_x & -\sin r_x & 0 \\ 0 & \sin r_x & \cos r_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \\
R_y &= \begin{bmatrix} \cos r_y & 0 & \sin r_y & 0 \\ 0 & 1 & 0 & 0 \\ -\sin r_y & 0 & \cos r_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \\
R_z &= \begin{bmatrix} \cos r_z & -\sin r_z & 0 & 0 \\ \sin r_z & \cos r_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \\
S &= \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \\
M &= T R_z R_y R_x S
\end{aligned}
\tag{15}
$$

In cases where composited primitives do not utilize the full set of transformation components—translation, rotation, and scale—we omit the respective term in the M formula. For uniform scaling, represented by a scalar scaling factor s, we set s_x, s_y, and s_z all equal to s.

6.2. Complete List of 3D Strokes

Utilizing various combinations of basic geometric shapes in unit space, along with transformations including translation, rotation, and scale, enables the creation of a diverse palette of 3D strokes. These strokes exhibit distinct geometric and aesthetic stylization. The full assortment of these 3D strokes is detailed in Tab. 3.

6.3. Spline Curves

6.3.1 Polynomial splines

In Sec. 3.2.2 of the main paper, we use three types of different polynomial curves that are commonly used in computer graphics. Specifically, we use the quadratic Bézier, cubic Bézier, and Catmull Rom spline, respectively. We provide the concrete definition of these curves below. All points defined here are vectors in the 3D scene space.

Quadratic Bézier Spline. A quadratic Bézier spline is defined by three control points, the start point $\mathbf{P}_0$, the end point $\mathbf{P}_2$, and middle point $\mathbf{P}_1$ that is also tangent to the $\mathbf{P}_0$ and $\mathbf{P}_2$. Note that the curve only blends toward but does not pass the middle point $\mathbf{P}_1$. The parametric form is given by:

$$
\mathbf{C}(t; \mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2) = (1-t)^2\mathbf{P}_0 + 2(1-t)t\mathbf{P}_1 + t^2\mathbf{P}_2 \tag{16}
$$

Cubic Bézier Spline. A cubic Bézier spline introduces an additional control point compared with the quadratic spline. This spline is defined by four points: the start point $\mathbf{P}_0$, two control points $\mathbf{P}_1$ and $\mathbf{P}_2$, and the end point $\mathbf{P}_3$. The curve starts at $\mathbf{P}_0$ and ends at $\mathbf{P}_3$, with $\mathbf{P}_1$ and $\mathbf{P}_2$ influencing its shape. The parametric equation of a cubic Bézier spline is:

$$
\begin{aligned}
\mathbf{C}(t; \mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3) = (1-t)^3\mathbf{P}_0 + 3(1-t)^2 t\mathbf{P}_1 \\
+ 3(1-t)t^2\mathbf{P}_2 + t^3\mathbf{P}_3
\end{aligned}
\tag{17}
$$

Catmull Rom Spline. The Catmull Rom spline is another form of cubic spline, notable for its ability to interpolate its control points. This spline is defined by a series of points, with the curve passing through each of these points except the first and last. One feature of the Catmull Rom spline is that the tangent at each point is determined by the line connecting the previous and next points, ensuring a smooth transition. In our implementation, we specifically use the centripetal Catmull-Rom spline, a variant of the standard Catmull-Rom spline. This type of spline is particularly advantageous for avoiding the issue of self-intersecting loops

Stroke Name	Base SDF	Translation	Rotation	Uniform Scale	Anisotropic Scale
Sphere	Unit Sphere	✓	✗	✓	✗
Ellipsoid	Unit Sphere	✓	✓	✗	✓
Axis-aligned Cube	Unit Cube	✓	✗	✓	✗
Oriented Cube	Unit Cube	✓	✓	✓	✗
Axis-aligned Box	Unit Cube	✓	✗	✗	✓
Oriented Box	Unit Cube	✓	✓	✗	✓
Round Cube	Unit RoundCube	✓	✓	✓	✗
Round Box	Unit RoundCube	✓	✓	✗	✓
Line	Unit Line	✓	✓	✓	✗
Triprism	Unit Triprism	✓	✓	✓	✗
Octahedron	Unit Octahedron	✓	✓	✓	✗
Tetrahedron	Unit Tetrahedron	✓	✓	✓	✗

Table 3. The basic shapes and transformations used in all 3D strokes.

in the curve, which are common in the uniform and chordal Catmull-Rom splines. The parametric form of the Catmull Rom spline, for a segment between $\mathbf{P}_1$ and $\mathbf{P}_2$, with $\mathbf{P}_0$ and $\mathbf{P}_3$ influencing the shape, is defined as:

$$\begin{aligned}\mathbf{C}(t;\mathbf{P}_0,\mathbf{P}_1,\mathbf{P}_2,\mathbf{P}_3) = \frac{1}{2}[&(2\mathbf{P}_1)\\ &+(-\mathbf{P}_0+\mathbf{P}_2)t\\ &+(2\mathbf{P}_0-5\mathbf{P}_1+4\mathbf{P}_2-\mathbf{P}_3)t^2\\ &+(-\mathbf{P}_0+3\mathbf{P}_1-3\mathbf{P}_2+\mathbf{P}_3)t^3]\end{aligned} \tag{18}$$

6.3.2 Nearest Point Finding

As mentioned in Sec. 3.2.2 of the main paper, we need to locate the nearest point on the spline in order to compute the SDF of the spline curve. This involves solving the following equation that finds the t value with the minimized distance to the query point $\mathbf{p} \in \mathbb{R}^3$ in a differential way:

$$t^* = \arg\min_t \|\mathbf{C}(t,\theta_s^{\text{curve}}) - \mathbf{p}\|_2,\ s.t.\ 0 \le t \le 1\ , \tag{19}$$

where θ_s^{curve} denotes the parameters of the spline curve. An analytical solution might exist for some specific formulations of the splines. However, for more versatility, we use a general approximation solution that can be adapted to any parametric spline curve in the main paper. $K+1$ samples are uniformly selected on the curve to form K line segments, and the distance of each line segment to the query point is calculated to find t^*. Assuming the line segment is given as $L(t;\mathbf{A},\mathbf{B}) = (1-t)\mathbf{A} + t\mathbf{B},\ t \in [0,1]$, where $\mathbf{A}$ and $\mathbf{B}$ are the two endpoints of the line segment, the distance from point $\mathbf{p}$ to this line segment L can be calculated as:

$$d_L(\mathbf{p},\mathbf{A},\mathbf{B}) = \|\mathbf{p} - L(t';\mathbf{A},\mathbf{B})\|_2 \tag{20}$$

where t' is the t value that gives the nearest point on the line segment:

$$t'(\mathbf{p},\mathbf{A},\mathbf{B}) = \min(\max(\frac{(\mathbf{p}-\mathbf{A})\cdot(\mathbf{B}-\mathbf{A})}{(\mathbf{B}-\mathbf{A})\cdot(\mathbf{B}-\mathbf{A})},0),1) \tag{21}$$

We then compute t^* using Algorithm 1. The computation complexity can be easily controlled by adjusting the number of K, where a higher K leads to a more accurate approximation but at the cost of higher computation.

Algorithm 1: Find the approximated nearest point on a given spline curve by uniformly sample K line segments on the curve.

Input: $\mathbf{C}$: parametric spline function defining the coordinates at distance $t \in [0,1]$.
θ_s^{curve}: parameters of the spline curve.
K: number of line segments used.
$\mathbf{p}$: coordinate of the query point.
Result: t^*: the t value of the nearest point on the spline curve.

```
t* ← 0;
t_start ← 0;
d_min ← ∞;
A ← C(t_start, θ_s^curve);
for i ← 1 to K do
    t_end ← i/K;
    B ← C(t_end, θ_s^curve);
    if d_L(p, A, B) < d_min then
        d_min = d_L(p, A, B);
        t* = t_start + (t_end − t_start) · t'(p, A, B);
    end
    t_start ← t_end;
    A ← B;
end
```

7. Implementation Details

We implement the stroke field using Pytorch [23] framework and implement the strokes using fused CUDA kernels to accelerate training and reduce GPU memory usage. We transform sampled coordinates to the canonical volume according to the scene bounding box and use the normalized scene coordinates as inputs to the stroke field. Like Zip-NeRF [2], we use a proposal network based on hash grid representation [20] to facilitate ray sampling. Specifically, for each ray of a pixel, we first sample 32 points using the proposal MLP to obtain sampling weights. We resample 32 points and compute the stroke field's density and color on each point, and use the same volumetric rendering formula in NeRF to acquire the final pixel color.

We train 15k steps using 500 strokes for scenes with a single object, and train 25k steps using 1000 strokes for face-forwarding scenes. We employ the AdamW [17] optimizer with betas $(0.9, 0.99)$, setting the learning rate to 0.01 and exponentially decays to 0.0003. We start with $k_\delta = 7$ and gradually decay it to $k_\delta = 1$ during training. Additionally, as brushes are progressively added to the scene, we start with 25% of the sampling points and gradually use all the sampling points when training reaches 80%. We set $\lambda_{color} = 1$, $\lambda_{mask} = 0.02$, $\lambda_{den\,reg} = 0.0001$, $\lambda_{err} = 0.1$, and $\lambda_{err\,reg} = 0.001$ in our training setup.

8. Quantitative comparison of strokes

In Sec. 4.1 of the main paper, we conduct a qualitative comparison between the visual effects of several selected 3D strokes. Additionally, this section includes a comprehensive quantitative analysis of all 3D strokes, as detailed in Tab. 4. The metrics are measured using 500 strokes on object scenes and 1000 strokes on face-forwarding scenes. As shown in the table, the ellipsoid stroke typically demonstrates superior fidelity in scene reconstruction, followed closely by the cubic Bézier curve.

Table 4. Averaged quantitative metrics of reconstruction results of different 3D strokes.

	PSNR↑	SSIM↑	LPIPS↓
Sphere	19.72	0.591	0.416
Ellipsoid	**21.78**	**0.687**	**0.283**
Cube	19.65	0.593	0.389
Axis-aligned Box	19.99	0.588	0.408
Oriented Box	20.66	0.637	0.321
Round Cube	20.17	0.623	0.380
Tetrahedron	20.30	0.625	0.372
Octahedron	20.09	0.614	0.380
Triprism	20.68	0.640	0.351
Line	20.74	0.641	0.347
Quadratic Bézier	21.32	0.676	0.311
Cubic Bézier	21.64	**0.687**	0.308
Catmull-Rom	21.47	0.675	0.324

9. Training setup of applications

9.1. Color Transfer

In Sec. 4.4 of the main paper, we transfer the color distribution from a reference style image to a trained stroke-based 3D scene. We adopt perceptual style loss, which extracts the gram matrix [8] at specific layers of a pre-trained VGG16 network [13], and computer the difference between the rendered RGB image and the target style image. Since computing style loss requires an image rather than individual pixels as input, we randomly render 32x32 chunks of original images under the given camera poses. We add this style loss with a weight $\lambda_{style} = 0.25$ to the total loss and fine-tune the color parameters of trained strokes for 5k iterations.

9.2. Text-driven scene drawing

In Sec. 4.4 of the main paper, we use the vision-language model CLIP [24] (ViT-B/32) to achieve scene drawing based on a given text prompt. Specifically, we minimize the loss between the text embedding of the given prompt and the image embedding of the rendered RGB image. When rendering the images, we sample camera poses in a circular path looking at the scene's origin with azimuth angle in $[0, 360]$ and elevation angle in $[40, 105]$, and render an image chunk of size 128×128. We consider the azimuth angle starting at zero as the front view and adjust the CLIP guidance scale for larger azimuth angles accordingly. This adjustment results in generation outcomes that are more coherent with the viewpoint.

For text guidance, we use the template "`realistic 3D rendering painting of [OBJECT]`", where `[OBJECT]` is substituted with the specific object description we aim to generate. Additionally, we discovered that incorporating a silhouette loss alongside the RGB loss significantly enhances the quality of the generated geometric shapes. This is achieved by generating another image embedding from the rendered opacity and encouraging lower loss between this silhouette image embedding with the text embedding.