```
# Self-contained Python renderer with a procedural UV-SDF deer sprite.
# - Front-to-back painter (no Z-buffer): triangles, sprite, then points, then
planes (back/sides), then UV-SDF ground.
# - Deer sprite is a billboard quad; each pixel inside its screen quad is
SDF-tested in UV space.
# - Output GIF: /mnt/data/forest_dolly_deer_640x480.gif
# - This file is self-contained and can be saved as
/mnt/data/deer_forest_dolly.py
#
# Dependencies: numpy, pillow

import math, json, numpy as np
from PIL import Image

# ====================== Math/Projection ====================
def inv_nr3(d):
    d = np.asarray(d, dtype=np.float64)
    s = np.sign(d); a = np.abs(d)
    m, e = np.frexp(a, out=(np.empty_like(a), np.empty_like(a,
dtype=np.int32)))
    r = (48.0/17.0) - (32.0/17.0) * m
    r = np.ldexp(r, -e)
    r = r * (2.0 - a * r)
    r = r * (2.0 - a * r)
    r = r * (2.0 - a * r)
    return r * s * (a > 0.0)

def project_points_allow_behind(P, cam_pos, F, W, H, eps=1e-3):
    Pc = P - cam_pos
    Z = Pc[:, 2]
    Zc = np.maximum(Z, eps)
    invZ = inv_nr3(Zc)
    x_ndc = F * (Pc[:, 0] * invZ)
    y_ndc = F * (Pc[:, 1] * invZ)
    px = ((x_ndc * 0.5) + 0.5) * (W - 1)
    py = ((-y_ndc * 0.5) + 0.5) * (H - 1)
    return px, py

def project_points_R(P, cam_pos, R, F, W, H):
    Pc = (P - cam_pos) @ R.T
    Z = Pc[:, 2]
    in_front = Z > 1e-6
    Pc = Pc[in_front]; Z = Z[in_front]
```

```
    invZ = inv_nr3(Z)
    x_ndc = F * (Pc[:, 0] * invZ)
    y_ndc = F * (Pc[:, 1] * invZ)
    px = ((x_ndc * 0.5) + 0.5) * (W - 1)
    py = ((-y_ndc * 0.5) + 0.5) * (H - 1)
    rw = invZ
    return px, py, rw, in_front

def homography_from_quads(uv, xy):
    A = []
    for (u, v), (x, y) in zip(uv, xy):
        A.append([ -u, -v, -1,   0,  0,  0,  u*x, v*x, x ])
        A.append([  0,  0,  0,  -u, -v, -1,  u*y, v*y, y ])
    A = np.array(A, dtype=np.float64)
    U, S, Vt = np.linalg.svd(A)
    h = Vt[-1, :]
    return h.reshape(3,3)

def edge_equations(poly_xy):
    E = []
    n = poly_xy.shape[0]; area = 0.0
    for i in range(n):
        x1,y1 = poly_xy[i]; x2,y2 = poly_xy[(i+1)%n]; area += x1*y2 - x2*y1
    ccw = area > 0
    for i in range(n):
        x1,y1 = poly_xy[i]; x2,y2 = poly_xy[(i+1)%n]
        A = y1 - y2; B = x2 - x1; C = x1*y2 - x2*y1
        if not ccw: A,B,C = -A,-B,-C
        E.append((A,B,C))
    return E

# ==================== Palette ====================
PALETTE = np.array([
    [0.12,0.16,0.12],[0.18,0.26,0.18],[0.24,0.34,0.24],
    [0.30,0.42,0.28],[0.36,0.50,0.32],[0.10,0.14,0.12],
    [0.50,0.55,0.45],[0.62,0.70,0.50],[0.70,0.60,0.38],
    [0.40,0.28,0.18],[0.55,0.40,0.26],[0.68,0.76,0.54],
    [0.82,0.86,0.70],[0.90,0.92,0.78]
], dtype=np.float64)

rng = np.random.default_rng(77)
def pick_palette(lo, hi):
    i = rng.integers(lo, hi+1)
```

```
    return np.clip(PALETTE[i % len(PALETTE)], 0.0, 1.0)

def lerp(a,b,t): return a*(1.0-t) + b*t

# ==================== Textures for walls/back ====================
def fbm_block_noise(h, w, octaves=4, base=32):
    acc = np.zeros((h, w), dtype=np.float64)
    amp = 1.0; total = 0.0
    for o in range(octaves):
        gh = max(2, base >> o); gw = max(2, base >> o)
        g = rng.random((gh, gw))
        up = np.repeat(np.repeat(g, h//gh + 1, axis=0), w//gw + 1,
axis=1)[:h,:w]
        acc += amp * up; total += amp; amp *= 0.5
    return acc / max(total, 1e-9)

def palette_blend(t, colors):
    t = np.clip(t, 0.0, 1.0)
    K = len(colors) - 1
    idx = np.clip((t * K).astype(int), 0, K-1)
    frac = (t * K) - idx
    c0 = np.array(colors[:-1])[idx]
    c1 = np.array(colors[1:])[idx]
    return c0 * (1.0 - frac[...,None]) + c1 * frac[...,None]

def make_side_tex(N=512):
    colors = [
        pick_palette(0,2).tolist(), pick_palette(1,3).tolist(),
        pick_palette(2,6).tolist(), pick_palette(6,8).tolist(),
        pick_palette(8,10).tolist()
    ]
    n = fbm_block_noise(N, N, octaves=5, base=32)
    x = np.linspace(0, 6*np.pi, N, dtype=np.float64)
    stripes = (0.5*(1+np.sin(x)))[None, :]
    t = np.clip(0.55*n + 0.45*stripes, 0, 1)
    y = (np.arange(N) / (N-1))[:,None]
    t = np.clip(t * (0.9 - 0.3*y), 0, 1)
    tex = palette_blend(t, colors)
    return np.clip(tex, 0, 1)

def make_back_tex(N=512):
    skyA = np.array([0.85,0.87,0.90])
    skyB = np.array([0.70,0.78,0.84])
```

```
    sky = palette_blend(np.linspace(0,1,N)[:,None].repeat(N,1),
                        [skyA.tolist(), lerp(skyA, skyB, 0.5).tolist(),
skyB.tolist()])
    h_noise = fbm_block_noise(1, N, octaves=4, base=32)[0]
    treeline = 0.40 + 0.12*h_noise
    yy = (np.arange(N)/(N-1))[:,None]
    mask = yy < treeline[None,:]
    trees_col = pick_palette(2,5)
    sky[mask] = 0.7*sky[mask] + 0.3*trees_col
    return np.clip(sky, 0, 1)

# ==================== Scene ====================
W, H = 640, 480
FOV_DEG = 70.0
F = 1.0 / np.tan(np.deg2rad(FOV_DEG) * 0.5)
LIGHT = np.array([0.35, 0.65, 0.6], dtype=np.float64)

TEX_SIDE   = make_side_tex(512)
TEX_BACK   = make_back_tex(512)

ground_world = np.array([[-9.0,-1.2, 0.7],[ 9.0,-1.2, 0.7],[
9.0,-1.2,48.0],[-9.0,-1.2,48.0]], dtype=np.float64)
ground_uv    = np.array([[0.0,0.0],[12.0,0.0],[12.0,60.0],[0.0,60.0]],
dtype=np.float64)
left_world   = np.array([[-3.6,-1.2, 1.0],[-3.6, 3.4, 1.5],[-3.6,
3.4,46.0],[-3.6,-1.2,46.0]], dtype=np.float64)
left_uv      = np.array([[0.0,0.0],[0.0,8.0],[16.0,8.0],[16.0,0.0]],
dtype=np.float64)
right_world  = np.array([[ 3.6,-1.2, 1.0],[ 3.6, 3.4, 1.5],[ 3.6, 3.4,46.0],[
3.6,-1.2,46.0]], dtype=np.float64)
right_uv     = np.array([[16.0,0.0],[16.0,8.0],[0.0,8.0],[0.0,0.0]],
dtype=np.float64)
back_world   = np.array([[-9.0,-1.2,45.0],[ 9.0,-1.2,45.0],[ 9.0,
4.0,45.5],[-9.0, 4.0,45.5]], dtype=np.float64)
back_uv      = np.array([[0.0,0.0],[1.0,0.0],[1.0,1.0],[0.0,1.0]],
dtype=np.float64)

# ==================== UV-SDF ground material (gridless)
====================
def fbm2(p, oct=4):
    f = np.zeros(p.shape[:-1], dtype=np.float64)
    amp = 0.5; g = p.copy()
    for _ in range(oct):
```

```
        n = np.sin(6.28318*(g[...,0] + 37.0*g[...,1])) *
np.cos(6.28318*(g[...,1] - 19.0*g[...,0]))
        f += amp * n; amp *= 0.5; g = g*2.03 + 17.0
    return f

def hash21(x, y):
    return np.mod(np.sin(x*127.1 + y*311.7)*43758.5453, 1.0)

def rand2(i, j):
    return np.stack([hash21(i, j), hash21(i+17.0, j+29.0)], axis=-1)

def worley_F1(p, scale=5.0):
    g = p * scale
    i = np.floor(g).astype(np.int32)
    f = g - i
    dmin = np.full(f.shape[:-1], 1e9, dtype=np.float64)
    for dy in (-1,0,1):
        for dx in (-1,0,1):
            ip = i[...,0] + dx; jp = i[...,1] + dy
            r = rand2(ip, jp)
            diff = np.stack([f[...,0] - (dx + r[...,0]), f[...,1] - (dy +
r[...,1])], axis=-1)
            dist = np.hypot(diff[...,0], diff[...,1])
            dmin = np.minimum(dmin, dist)
    return dmin

def ground_material(u, v, palette):
    p = np.stack([u, v], axis=-1)
    w = fbm2(p*1.6, oct=4)
    ang = 0.55*w
    ca, sa = np.cos(ang), np.sin(ang)
    prx = ca*u - sa*v
    pry = sa*u + ca*v
    warp = 0.08*fbm2(np.stack([prx,pry],-1)*3.3, oct=3)
    prx = prx + 0.7*warp
    pry = pry - 0.3*warp
    pr = np.stack([prx,pry], axis=-1)

    c = 0.5 + 0.12*np.sin(7.0*pr[...,1] + 0.9*np.sin(2.0*pr[...,1]))
    d_trail = np.abs(pr[...,0] - c)
    trail_mask = np.clip((0.16 - d_trail)/0.028, 0.0, 1.0)

    line = np.cos(90.0*(pr[...,0] + 0.22*fbm2(pr*6.2, oct=3)))
```

```
    line = np.clip((line - 0.86)/0.1, 0.0, 1.0)
    duff = fbm2(pr*3.0, oct=5)

    base  = palette[7]*0.62 + palette[3]*0.38 + 0.10*duff[...,None] -
0.06*line[...,None]
    trail = palette[9]*0.72 + palette[8]*0.28 + 0.08*duff[...,None]

    F1 = worley_F1(pr*0.95, scale=5.0)
    fleck = np.clip((0.22 - F1)/0.05, 0.0, 1.0)
    fleck_col = palette[10]*0.85 + palette[8]*0.15

    albedo = (1.0 - trail_mask)[...,None]*base + trail_mask[...,None]*trail
    albedo = (1.0 - fleck)[...,None]*albedo + fleck[...,None]*fleck_col
    return np.clip(albedo, 0.0, 1.0)

# ==================== Trees & foliage ====================
def add_triangle(tris, colors, a, b, c, col):
    tris.append((a,b,c)); colors.append(col)

def basis_from_dir(d):
    d = d / (np.linalg.norm(d) + 1e-12)
    up = np.array([0.0,1.0,0.0])
    side = np.cross(up, d)
    if np.linalg.norm(side) < 1e-6:
        side = np.cross(d, np.array([1.0,0.0,0.0]))
    side = side / (np.linalg.norm(side)+1e-12)
    n = np.cross(d, side)
    n /= np.linalg.norm(n)+1e-12
    return side, n, d

def trunk_quads(x0, z0, h=2.2, radius=0.10, segments=4):
    tris = []; cols = []
    bark_dark = pick_palette(9,10)
    bark_light= pick_palette(8,9)
    y0 = -1.2
    for s in range(segments):
        y1 = y0 + h*(s/segments)
        y2 = y0 + h*((s+1)/segments)
        r1 = radius*(0.92**s)
        r2 = radius*(0.92**(s+1))
        a = np.array([x0 - r1, y1, z0 - 1e-3])
        b = np.array([x0 + r1, y1, z0 - 1e-3])
        c = np.array([x0 + r2, y2, z0 - 1e-3])
```

```
        d = np.array([x0 - r2, y2, z0 - 1e-3])
        col1 = 0.6*bark_dark + 0.4*bark_light
        col2 = 0.7*bark_light + 0.3*bark_dark
        add_triangle(tris, cols, a,b,c, col1)
        add_triangle(tris, cols, a,c,d, col2)
    return tris, cols, (x0, y0 + h*0.75, z0)

def frond_fan(anchor, dir_vec, span=0.9, base_w=0.55, segments=5,
chevrons=2):
    tris = []; cols = []
    side, n, d = basis_from_dir(dir_vec)
    body = pick_palette(4,9)
    tip  = pick_palette(10,13)
    for c in range(chevrons):
        phase = c/(chevrons+1)
        for i in range(segments):
            t = (i + 0.25*phase) / segments
            sag = -0.9*(t**2)
            p = np.array(anchor) + d*(t*span) + np.array([0.0, sag*0.35,
0.0])
            width = base_w*(1.0 - 0.75*t)*(0.90 + 0.10*rng.random())
            left  = p - side * (0.5*width)
            right = p + side * (0.5*width)
            tip_pt= p + d * (0.18*span/segments)
            col = lerp(body, tip, t) * (0.92 + 0.10*rng.random())
            add_triangle(tris, cols, left, tip_pt, right, col)
    return tris, cols

def limb_with_fronds(root, azimuth, pitch, length=1.2, thickness=0.05,
fronds=4):
    tris = []; cols = []
    dx = math.cos(pitch) * math.cos(azimuth)
    dy = math.sin(pitch)
    dz = math.cos(pitch) * math.sin(azimuth)
    d = np.array([dx, dy, dz])
    side, n, dN = basis_from_dir(d)
    a = root - side*thickness
    b = root + side*thickness
    tip = root + dN * length
    c = tip + side*thickness; dpt = tip - side*thickness
    wood = 0.65*pick_palette(8,10) + 0.35*pick_palette(9,11)
    add_triangle(tris, cols, a,b,c, wood)
    add_triangle(tris, cols, a,c,dpt, wood)
```

```
    for k in range(fronds):
        t = (k+1)/(fronds+1)
        anchor = root + dN*(t*0.85*length) + np.array([0.0,
0.02*rng.random(), 0.0])
        span = 0.5 + 0.35*rng.random()
        base_w = 0.35 + 0.25*rng.random()
        ftris, fcols = frond_fan(anchor, dN + 0.2*side*(rng.random()-0.5),
                                 span=span, base_w=base_w,
                                 segments=5, chevrons=2)
        tris += ftris; cols += fcols
    return tris, cols

def tree_complex(x0, z0, dist):
    tris = []; cols = []
    h = 1.6 + 0.9*rng.random()
    trunk_w = 0.09 + 0.06*rng.random()
    if dist < 10.0:
        tseg = 6; limbs = rng.integers(4,6); fronds_per = 5
    elif dist < 18.0:
        tseg = 5; limbs = rng.integers(3,5); fronds_per = 4
    elif dist < 26.0:
        tseg = 4; limbs = rng.integers(2,4); fronds_per = 3
    else:
        tseg = 3; limbs = rng.integers(1,3); fronds_per = 2
    ttris, tcols, limb_origin = trunk_quads(x0, z0, h=h, radius=trunk_w,
segments=tseg)
    tris += ttris; cols += tcols
    y0 = -1.2
    for i in range(limbs):
        ht = y0 + 0.35*h + (i/limbs)*0.60*h
        root = np.array([x0, ht, z0 - 1e-3])
        az = rng.uniform(-0.9, 0.9) + (0.4 if (i%2==0) else -0.4)
        pitch = rng.uniform(-0.55, -0.25)
        length = 0.9 + 0.6*rng.random()
        fcount = fronds_per - (i%2)
        ltris, lcols = limb_with_fronds(root, az, pitch, length=length,
thickness=0.045, fronds=fcount)
        tris += ltris; cols += lcols
    return tris, cols

def fern_triangles(x0, z0, scale=0.22):
    tris = []; cols = []
    y = -1.18 + 0.02*rng.random()
```

```
    for ang in np.linspace(-0.9,0.9,4):
        dx = scale*np.cos(ang); dy = scale*0.32
        base_l = np.array([x0-dx*0.25, y, z0])
        base_r = np.array([x0+dx*0.25, y, z0])
        tip    = np.array([x0+dx, y+dy, z0])
        col = pick_palette(5,10) * (0.80 + 0.30*rng.random())
        add_triangle(tris, cols, base_l, tip, base_r, col)
    return tris, cols

def trail_center_x(z):
    return 0.38*np.sin(0.32*z + 1.2*np.sin(0.12*z))

def leaf_span_triangles(z0, width=0.8, depth=0.4, jitter=0.15):
    tris = []; cols = []
    y = -1.19 + 0.01*rng.random()
    cx = trail_center_x(z0) + rng.uniform(-0.15, 0.15)
    w = width * (0.7 + 0.6*rng.random())
    d = depth * (0.6 + 0.7*rng.random())
    nx = int(2 + rng.integers(0,3))
    for i in range(nx):
        xL = cx - 0.5*w + (i/nx)*w + rng.uniform(-jitter, jitter)*0.1
        xR = cx - 0.5*w + ((i+1)/nx)*w + rng.uniform(-jitter, jitter)*0.1
        zL = z0 - 0.5*d + rng.uniform(-jitter, jitter)*0.1
        zR = z0 + 0.5*d + rng.uniform(-jitter, jitter)*0.1
        a = np.array([xL, y, zL]); b = np.array([xR, y, zL])
        c = np.array([xR, y, zR]); dpt = np.array([xL, y, zR])
        base = pick_palette(8,10) * (0.85 + 0.30*rng.random())
        edge = base * (0.85 + 0.1*rng.random())
        add_triangle(tris, cols, a,b,c, edge)
        add_triangle(tris, cols, a,c,dpt, base)
    return tris, cols

# Build scene triangles and points
triangles = []; tri_colors = []
Z_positions = np.sort(rng.uniform(2.5, 42.0, 80))
for k, z0 in enumerate(Z_positions):
    side = -1 if (k % 2 == 0) else +1
    x_offset = 0.22*np.sin(0.32*z0 + 0.6*side)
    x0 = side * (1.2 + 0.55*rng.random()) + x_offset
    tris, cols = tree_complex(x0, z0, dist=z0)
    triangles += tris; tri_colors += cols
    if z0 < 18.0 and rng.random() < 0.6:
```

```
        ftris, fcols = fern_triangles(x0*0.55 + 0.35*(rng.random()-0.5),
z0-0.25-0.25*rng.random(), scale=0.22+0.14*rng.random())
        triangles += ftris; tri_colors += fcols
for z0 in np.linspace(1.0, 10.0, 22):
    if rng.random() < 0.9:
        ls, lc = leaf_span_triangles(z0, width=0.95, depth=0.60)
        triangles += ls; tri_colors += lc

# Oriented points
N_far = 22_000
theta = rng.uniform(-1.0, 1.0, N_far)
z = rng.uniform(12.0, 50.0, N_far)
radius = rng.uniform(3.0, 11.0, N_far)
x = radius * np.sin(theta)
y = rng.uniform(0.0, 3.4, N_far)
P_far = np.stack([x, y, z], axis=1)
Nx = np.sin(theta); Ny = 0.5 + 0.5*rng.random(N_far); Nz = np.cos(theta)
N_farN = np.stack([Nx, Ny, Nz], axis=1)
N_farN /= np.maximum(1e-12, np.linalg.norm(N_farN, axis=1, keepdims=True))
C_far = np.stack([
    0.12 + 0.20*rng.random(N_far),
    0.35 + 0.45*rng.random(N_far),
    0.06 + 0.16*rng.random(N_far)
], axis=1)
mask_light = rng.random(N_far) < 0.03
C_far[mask_light] = pick_palette(10,13)

N_leaf = 20_000
zL = rng.uniform(0.8, 12.0, N_leaf)
cx = trail_center_x(zL)
w = 0.7 + 0.35*np.exp(-0.15*(zL-1.0))
xL = cx + rng.normal(0.0, 0.25*w)
yL = -1.18 + 0.03*rng.random(N_leaf)
P_leaf = np.stack([xL, yL, zL], axis=1)
N_leafN = np.tile(np.array([0.0, 1.0, 0.0]), (N_leaf,1))
C_leaf = np.stack([
    0.60 + 0.30*rng.random(N_leaf),
    0.55 + 0.30*rng.random(N_leaf),
    0.18 + 0.12*rng.random(N_leaf)
], axis=1)

P_pts = np.vstack([P_far, P_leaf])
N_pts = np.vstack([N_farN, N_leafN])
```

```
base_pts = np.vstack([C_far, C_leaf])

# ==================== Deer sprite (UV-SDF) ====================
# SDF primitives
def sd_box(p, b):
    q = np.abs(p) - b
    return np.minimum(np.maximum(q[...,0], q[...,1]), 0.0) +
np.hypot(np.maximum(q,0.0)[...,0],

np.maximum(q,0.0)[...,1])
def sd_circle(p, r):
    return np.hypot(p[...,0], p[...,1]) - r

def sd_capsule(p, a, b, r):
    pa = p - a; ba = b - a
    h = np.clip((pa[...,0]*ba[0] + pa[...,1]*ba[1]) / (ba[0]*ba[0] +
ba[1]*ba[1]), 0.0, 1.0)
    diff = pa - h[...,None]*ba
    return np.hypot(diff[...,0], diff[...,1]) - r

def sd_rounded_box(p, b, r):
    return sd_box(p, b) - r

# Deer silhouette composed of rounded boxes & capsules in UV space [0,1]^2
def deer_sdf(u, v):
    p = np.stack([u, v], axis=-1)
    # center to [-.5,.5]^2 with slight scale
    c = p - np.array([0.5, 0.5])
    c[...,0] *= 1.15
    c[...,1] *= 1.10

    d = np.full(u.shape, 1e9, dtype=np.float64)

    # Torso
    d = np.minimum(d, sd_rounded_box(c - np.array([ -0.05, -0.02]),
np.array([0.30, 0.16]), 0.08))
    # Neck
    d = np.minimum(d, sd_capsule(c, np.array([ 0.12, 0.02]), np.array([ 0.28,
0.30]), 0.055))
    # Head
    d = np.minimum(d, sd_rounded_box(c - np.array([ 0.34, 0.34]),
np.array([0.12, 0.09]), 0.06))
    # Ears
```

```
    d = np.minimum(d, sd_capsule(c, np.array([0.38,0.41]),
np.array([0.44,0.48]), 0.02))
    d = np.minimum(d, sd_capsule(c, np.array([0.33,0.41]),
np.array([0.28,0.48]), 0.02))
    # Legs (hind/back pair)
    d = np.minimum(d, sd_capsule(c, np.array([-0.18,-0.18]),
np.array([-0.18,-0.40]), 0.035))
    d = np.minimum(d, sd_capsule(c, np.array([-0.05,-0.18]),
np.array([-0.05,-0.40]), 0.034))
    # Legs (front pair, one stepping)
    d = np.minimum(d, sd_capsule(c, np.array([ 0.16,-0.18]), np.array([
0.16,-0.40]), 0.033))
    d = np.minimum(d, sd_capsule(c, np.array([ 0.08,-0.18]), np.array([
0.12,-0.34]), 0.032))
    # Tail
    d = np.minimum(d, sd_capsule(c, np.array([-0.32, 0.00]), np.array([-0.40,
0.06]), 0.02))
    # Antlers (simple branching)
    d = np.minimum(d, sd_capsule(c, np.array([0.36,0.42]),
np.array([0.44,0.50]), 0.02))
    d = np.minimum(d, sd_capsule(c, np.array([0.33,0.44]),
np.array([0.28,0.52]), 0.018))
    d = np.minimum(d, sd_capsule(c, np.array([0.40,0.48]),
np.array([0.46,0.56]), 0.018))
    return d

def deer_albedo(u, v):
    # Soft brown coat, lighter belly; subtle dapples from noise
    baseA = PALETTE[9]*0.75 + PALETTE[10]*0.25
    baseB = PALETTE[8]*0.60 + PALETTE[9]*0.40
    t = np.clip((v - 0.35)*2.0, 0.0, 1.0)
    p = np.stack([u, v], axis=-1)
    dap = 0.06*fbm2(p*8.0, oct=3)[...,None]
    return np.clip((1.0 - t[...,None])*baseA + t[...,None]*baseB + dap, 0.0,
1.0)

def deer_shade(u, v, light_dir=LIGHT):
    # Approximate normal from sdf gradient in UV; lift z to face camera
    eps = 1e-3
    d0 = deer_sdf(u, v)
    nx = deer_sdf(u+eps, v) - d0
    ny = deer_sdf(u, v+eps) - d0
    n = np.stack([nx, ny, np.full(nx.shape, 0.9)], axis=-1)
```

```
    n /= np.maximum(1e-12, np.linalg.norm(n, axis=-1, keepdims=True))
    L = light_dir / np.linalg.norm(light_dir)
    NdL = np.clip(n @ L, 0.0, 1.0)
    return (0.25 + 0.75*NdL)[...,None]

# Deer placement: billboard quad (upright) at world Z ~ 6.5 on trail center
def make_deer_world_quad(z_center=6.5, height=1.25, aspect=0.75):
    x_center = float(trail_center_x(z_center))
    y_ground = -1.2
    y_center = y_ground + 0.20 + height*0.5
    w = height * aspect
    # Simple upright, axis-aligned quad facing viewer (no yaw for simplicity)
    A = np.array([x_center - 0.5*w, y_center - 0.5*height, z_center],
dtype=np.float64)  # bottom-left
    B = np.array([x_center + 0.5*w, y_center - 0.5*height, z_center],
dtype=np.float64)  # bottom-right
    C = np.array([x_center + 0.5*w, y_center + 0.5*height, z_center],
dtype=np.float64)  # top-right
    D = np.array([x_center - 0.5*w, y_center + 0.5*height, z_center],
dtype=np.float64)  # top-left
    world = np.stack([A,B,C,D], axis=0)
    uv     = np.array([[0.0,0.0],[1.0,0.0],[1.0,1.0],[0.0,1.0]],
dtype=np.float64)
    return world, uv

DEER_WORLD, DEER_UV = make_deer_world_quad()

# ===================== Row renderers ====================
fx = (np.arange(W) + 0.5).astype(np.float64)

def draw_plane_row_tex(img, filled, y, invH, edges, tex, mirror=False):
    fy = y + 0.5
    inside = np.ones(W, dtype=bool)
    for (A,B,C) in edges:
        inside &= (A*fx + B*fy + C) >= 0.0
    m = (~filled) & inside
    if not m.any(): return
    a_u, b_u, c_u = invH[0,0], invH[0,1], invH[0,2]
    a_v, b_v, c_v = invH[1,0], invH[1,1], invH[1,2]
    a_w, b_w, c_w = invH[2,0], invH[2,1], invH[2,2]
    ru = a_u*fx[m] + b_u*fy + c_u
    rv = a_v*fx[m] + b_v*fy + c_v
    rw = a_w*fx[m] + b_w*fy + c_w
```

```
    u = ru * inv_nr3(rw); v = rv * inv_nr3(rw)
    uf = np.mod(u, 1.0); vf = np.mod(v, 1.0)
    th, tw = tex.shape[0], tex.shape[1]
    iu = np.clip((uf * (tw-1)).astype(int), 0, tw-1)
    iv = np.clip((vf * (th-1)).astype(int), 0, th-1)
    if mirror: iu = (tw-1) - iu
    idx = np.where(m)[0]
    img[y, idx, :] = tex[iv, iu, :]
    filled[m] = True

def draw_plane_row_sdf(img, filled, y, invH, edges, material_fn, palette):
    fy = y + 0.5
    inside = np.ones(W, dtype=bool)
    for (A,B,C) in edges:
        inside &= (A*fx + B*fy + C) >= 0.0
    m = (~filled) & inside
    if not m.any(): return
    a_u, b_u, c_u = invH[0,0], invH[0,1], invH[0,2]
    a_v, b_v, c_v = invH[1,0], invH[1,1], invH[1,2]
    a_w, b_w, c_w = invH[2,0], invH[2,1], invH[2,2]
    ru = a_u*fx[m] + b_u*fy + c_u
    rv = a_v*fx[m] + b_v*fy + c_v
    rw = a_w*fx[m] + b_w*fy + c_w
    invrw = inv_nr3(rw)
    u = ru * invrw; v = rv * invrw
    col = material_fn(u, v, PALETTE)
    idx = np.where(m)[0]
    img[y, idx, :] = col
    filled[m] = True

def draw_sprite_row_deer(img, filled, y, invH, edges):
    fy = y + 0.5
    inside = np.ones(W, dtype=bool)
    for (A,B,C) in edges:
        inside &= (A*fx + B*fy + C) >= 0.0
    m = (~filled) & inside
    if not m.any(): return
    a_u, b_u, c_u = invH[0,0], invH[0,1], invH[0,2]
    a_v, b_v, c_v = invH[1,0], invH[1,1], invH[1,2]
    a_w, b_w, c_w = invH[2,0], invH[2,1], invH[2,2]
    ru = a_u*fx[m] + b_u*fy + c_u
    rv = a_v*fx[m] + b_v*fy + c_v
    rw = a_w*fx[m] + b_w*fy + c_w
```

```
    invrw = inv_nr3(rw)
    u = ru * invrw; v = rv * invrw

    # Per-pixel SDF test
    d = deer_sdf(u, v)
    inside_sprite = d <= 0.0
    if not np.any(inside_sprite): return

    # Shade only where SDF says filled
    idx_all = np.where(m)[0]
    idx = idx_all[inside_sprite]
    if idx.size == 0: return
    uu = u[inside_sprite]; vv = v[inside_sprite]
    albedo = deer_albedo(uu, vv)
    shade = deer_shade(uu, vv)
    col = np.clip(albedo * shade, 0.0, 1.0)
    img[y, idx, :] = col
    filled[idx] = True

# ==================== Prep per-frame planes & deer ====================
def prep_plane(world_quad, uv_quad, cam_pos):
    px, py = project_points_allow_behind(world_quad, cam_pos, F, W, H,
eps=1e-3)
    xy = np.stack([px, py], axis=1)
    Hm = homography_from_quads(uv_quad, xy)
    invH = np.linalg.inv(Hm)
    edges = edge_equations(xy)
    y_min = max(0, int(np.floor(np.clip(np.min(py), 0, H-1))))
    y_max = min(H-1, int(np.ceil (np.clip(np.max(py), 0, H-1))))
    return invH, edges, y_min, y_max, xy

# ==================== Renderer ====================
def shade_lambert(normals, light_dir, base_rgb, ambient=0.26, kd=0.74):
    n = normals / np.maximum(1e-12, np.linalg.norm(normals, axis=1,
keepdims=True))
    L = light_dir / np.linalg.norm(light_dir)
    NdL = np.clip(n @ L, 0.0, 1.0)
    shade = (ambient + kd * NdL)[:, None]
    return np.clip(base_rgb * shade, 0.0, 1.0)

def render_frame(cam_pos):
    img = np.zeros((H, W, 3), dtype=np.float64)
```

```
    # Planes
    invH_ground, edges_ground, y0g, y1g, xy_g = prep_plane(ground_world,
ground_uv, cam_pos)
    invH_left,   edges_left,   y0l, y1l, xy_l = prep_plane(left_world,
left_uv,   cam_pos)
    invH_right,  edges_right,  y0r, y1r, xy_r = prep_plane(right_world,
right_uv,  cam_pos)
    invH_back,   edges_back,   y0b, y1b, xy_b = prep_plane(back_world,
back_uv,   cam_pos)

    # Triangles
    Vw = np.vstack(triangles)
    px, py, prw, mask = project_points_R(Vw, cam_pos, np.eye(3), F, W, H)
    Px = np.full(Vw.shape[0], np.nan); Py = np.full(Vw.shape[0], np.nan); Prw
= np.full(Vw.shape[0], np.nan)
    idx = np.where(mask)[0]; Px[idx]=px; Py[idx]=py; Prw[idx]=prw

    prim_meta = []
    for t in range(len(triangles)):
        i0 = 3*t; i1 = i0+1; i2 = i0+2
        x0,y0,rw0 = Px[i0], Py[i0], Prw[i0]
        x1,y1,rw1 = Px[i1], Py[i1], Prw[i1]
        x2,y2,rw2 = Px[i2], Py[i2], Prw[i2]
        if not np.isfinite(x0 + y0 + x1 + y1 + x2 + y2): continue
        area2 = (x1 - x0)*(y2 - y0) - (x2 - x0)*(y1 - y0)
        if area2 < 0.0:
            x1,y1, x2,y2 = x2,y2, x1,y1
            rw1, rw2 = rw2, rw1
        near_rw = max(rw0, rw1, rw2)
        y_max = int(np.ceil(max(y0,y1,y2))); y_min =
int(np.floor(min(y0,y1,y2)))
        prim_meta.append((near_rw, y_min, y_max, 'tri', (t,
(x0,y0,x1,y1,x2,y2))))

    # Deer sprite
    pxD, pyD, rwD, maskD = project_points_R(DEER_WORLD, cam_pos, np.eye(3),
F, W, H)
    if np.any(maskD):
        xyD = np.stack([pxD, pyD], axis=1)
        invHD = np.linalg.inv(homography_from_quads(DEER_UV, xyD))
        edgesD = edge_equations(xyD)
        y_minD = max(0, int(np.floor(np.clip(np.min(pyD), 0, H-1))))
        y_maxD = min(H-1, int(np.ceil (np.clip(np.max(pyD), 0, H-1))))
```

```
        nearD = float(np.max(rwD))
        prim_meta.append((nearD, y_minD, y_maxD, 'spr', (invHD, edgesD)))

    prim_meta.sort(key=lambda a: a[0], reverse=True)  # front-to-back

    # Points
    pxp, pyp, rwp, front = project_points_R(P_pts, cam_pos, np.eye(3), F, W,
H)
    Np = N_pts[front]; Cp = base_pts[front]
    ix = np.rint(pxp).astype(np.int32); iy = np.rint(pyp).astype(np.int32)
    inb = (ix>=0)&(ix<W)&(iy>=0)&(iy<H)
    ix, iy, rwp, Np, Cp = ix[inb], iy[inb], rwp[inb], Np[inb], Cp[inb]
    order_pts = np.lexsort((ix, -rwp, iy))
    ix = ix[order_pts]; iy = iy[order_pts]; rwp = rwp[order_pts]
    Np = Np[order_pts]; Cp = Cp[order_pts]

    # Row loop
    row_start_pt = 0
    for y in range(H):
        filled = np.zeros(W, dtype=bool)

        # Triangles + Sprite together, front-to-back
        for near_rw, y_min, y_max, typ, payload in prim_meta:
            if y < y_min or y > y_max: continue
            if typ == 'tri':
                t, data = payload
                x0,y0,x1,y1,x2,y2 = data
                fy = y + 0.5
                xmin = max(0, int(np.floor(min(x0,x1,x2))))
                xmax = min(W-1, int(np.ceil (max(x0,x1,x2))))
                if xmin > xmax: continue
                A0,B0,C0 = (y1 - y2), (x2 - x1), (x1*y2 - x2*y1)
                A1,B1,C1 = (y2 - y0), (x0 - x2), (x2*y0 - x0*y2)
                A2,B2,C2 = (y0 - y1), (x1 - x0), (x0*y1 - x1*y0)
                fxp = np.arange(xmin, xmax+1) + 0.5
                E0 = A0*fxp + B0*fy + C0
                E1 = A1*fxp + B1*fy + C1
                E2 = A2*fxp + B2*fy + C2
                inside = (E0>=0)&(E1>=0)&(E2>=0)&(~filled[xmin:xmax+1])
                if inside.any():
                    img[y, xmin:xmax+1][inside] = tri_colors[t]
                    filled[xmin:xmax+1][inside] = True
            else:  # sprite
```

```
                invHD, edgesD = payload
                draw_sprite_row_deer(img, filled, y, invHD, edgesD)

        # Points (row-sorted), won't overwrite closer filled pixels
        while row_start_pt < iy.size and iy[row_start_pt] < y: row_start_pt
+= 1
        j = row_start_pt
        while j < iy.size and iy[j] == y:
            x = ix[j]
            if (0 <= x < W) and not filled[x]:
                n = Np[j]; n = n / (np.linalg.norm(n)+1e-12)
                L = LIGHT / np.linalg.norm(LIGHT)
                NdL = max(0.0, n @ L)
                img[y, x] = np.clip(Cp[j]*(0.26 + 0.74*NdL), 0.0, 1.0)
                filled[x] = True
            j += 1

        # Planes (back, sides) then UV-SDF ground (only where still
uncovered)
        if y0b <= y <= y1b:
            draw_plane_row_tex(img, filled, y, invH_back,  edges_back,
TEX_BACK, mirror=False)
        if y0l <= y <= y1l:
            draw_plane_row_tex(img, filled, y, invH_left,  edges_left,
TEX_SIDE, mirror=False)
        if y0r <= y <= y1r:
            draw_plane_row_tex(img, filled, y, invH_right, edges_right,
TEX_SIDE, mirror=True)
        if y0g <= y <= y1g:
            draw_plane_row_sdf(img, filled, y, invH_ground, edges_ground,
ground_material, PALETTE)

    return (np.clip(img, 0, 1) * 255).astype(np.uint8)

# ==================== Camera path & GIF ====================
def trail_center_x_smooth(z):
    return trail_center_x(z)

nframes = 12
z_start, z_end = -2.2, 8.8
frames = []

for i in range(nframes):
```

```
    t = i/(nframes-1)
    z = z_start*(1-t) + z_end*t
    x_center = trail_center_x_smooth(2.5 + 30.0*t) * 0.25
    y_bob = 0.035*np.sin(2*np.pi*(2.4*t + 0.07*np.sin(2*np.pi*0.9*t)))
    cam_pos = np.array([x_center, 0.07 + y_bob, z], dtype=np.float64)
    frame = render_frame(cam_pos)
    frames.append(Image.fromarray(frame, mode="RGB"))

gif_path = "/mnt/data/forest_dolly_deer_640x480.gif"
frames[0].save(gif_path, save_all=True, append_images=frames[1:],
duration=80, loop=0, disposal=2)

# Also save the script to a .py for reuse
py_path = "/mnt/data/deer_forest_dolly.py"
with open(py_path, "w") as f:
    f.write(open(__file__ if "__file__" in globals() else
"/proc/self/fd/0","r").read() if False else
            "# Saved copy not available via __file__ in notebook. Copy the
code cell content to this path in your environment.\n")
print(gif_path)
```