import tensorflow as tf

import numpy as np

import data\_handling

import models

def\_dtype=np.float32

###### Ray generation

def get\_rays\_zoom(H, W, focal, c2w, factor):

    """

    Generate rays with a zoom factor.

    Generates a gridded set of rays following a desired grid width, grid height, and camera parameters.

    Parameters:

    H (int): Height ray grid

    W (int): Width of ray grid

    focal (float): Focal length of camera

    c2w (Tensor[4,4]): Camera to world matrix

    factor (float): Zoom factor (smaller than 1 = zoom)

    Returns:

    rays\_o (Tensor[H\*W, 3]): 3D origin of rays in absolute ref. frame

    rays\_d (Tensor[H\*W, 3]): 3D direction of rays in absolute ref. frame

    """

    # Ray origins in camera reference frame

    i, j = tf.meshgrid(tf.linspace(0.0, W\*factor, int(W)), tf.linspace(0.0, H\*factor, int(H)), indexing='xy')

    # Ray directions in camera reference frame

    dirs = tf.stack([(i-W\*factor\*.5)/focal, -(j-H\*factor\*.5)/focal, -tf.ones\_like(i)], -1)

    # Transform rays to absolute ref. frame using c2w matrix

    rays\_d = tf.reduce\_sum(dirs[..., np.newaxis, :] \* c2w[:3,:3], -1)

    rays\_o = tf.broadcast\_to(c2w[:3,-1], tf.shape(rays\_d))

    return tf.reshape(rays\_o, shape=[-1,3]), tf.reshape(rays\_d, shape=[-1,3])

def generate\_train\_rays(dataset, arg\_dict):

    """

    Generate complete ray set based on training images in a dataset.

    A ray is a 3d origin, a 3d direction, a 3d color value, and optionally 2d light and 2d view directions.

    Parameters:

    dataset (dict): dataset as specified in 'data\_handling.generate\_dataset'

    arg\_dict (dict): config variables

    Returns:

    all\_train(dict): set of all rays

    """

    use\_view\_dirs = arg\_dict['model.ins.views']

    use\_light\_dirs = arg\_dict['model.ins.light']

    el\_adj = arg\_dict['rend.unzoom']

    all\_rays\_o, all\_rays\_d, all\_values = [], [], []

    if use\_view\_dirs:

        all\_view\_dirs = []

    if use\_light\_dirs:

        all\_light\_dirs = []

    for view\_i in range(len(dataset['train\_imgs'])):

        H, W, nbands = dataset['train\_imgs'][view\_i].shape

        focals, poses = dataset['train\_focals'], dataset['train\_poses']

        if el\_adj:

            el = dataset['train\_view\_dirs'][view\_i][0,1]

            # Special case to handle satellite images, unzoom by 1/sin(elevation) to compensate

            rays\_o, rays\_d = get\_rays\_zoom(H, W, focals[view\_i], poses[view\_i], 1/np.sin(el))

        else:

            rays\_o, rays\_d = get\_rays\_zoom(H, W, focals[view\_i], poses[view\_i], 1.0)

        rays\_o = tf.reshape(rays\_o, [H, W, 3])

        rays\_d = tf.reshape(rays\_d, [H, W, 3])

        values = dataset['train\_imgs'][view\_i]

        all\_rays\_o.append(rays\_o)

        all\_rays\_d.append(rays\_d)

        all\_values.append(values)

        if use\_view\_dirs:

            all\_view\_dirs.append(tf.ones([rays\_o.shape[0],rays\_o.shape[1],1])@dataset['train\_view\_dirs'][view\_i])

        if use\_light\_dirs:

            all\_light\_dirs.append(tf.ones([rays\_o.shape[0],rays\_o.shape[1],1])@dataset['train\_light\_dirs'][view\_i])

    all\_values = tf.reshape(tf.convert\_to\_tensor(all\_values), [-1, nbands])

    all\_rays\_o = tf.reshape(tf.convert\_to\_tensor(all\_rays\_o), [-1, 3])

    all\_rays\_d = tf.reshape(tf.convert\_to\_tensor(all\_rays\_d), [-1, 3])

    all\_train = {'rays\_o':all\_rays\_o, 'rays\_d':all\_rays\_d, 'values':all\_values}

    if use\_view\_dirs:

        all\_train['view\_dirs'] = tf.reshape(tf.convert\_to\_tensor(all\_view\_dirs), [-1, 2])

    if use\_light\_dirs:

        all\_train['light\_dirs'] = tf.reshape(tf.convert\_to\_tensor(all\_light\_dirs), [-1, 2])

    return all\_train

def generate\_train\_light\_correction\_rays(dataset, arg\_dict):

    """

    Generate ray set for solar correction based on the solar angles from the training data set.

    Here no values are required, but light directions are necessary for solar correction.

    Parameters:

    dataset (dict): dataset as specified in 'data\_handling.generate\_dataset'

    arg\_dict (dict): config variables

    Returns:

    all\_train(dict): origin, direction, light direction of solar correction rays.

    """

    light\_df = arg\_dict['train.shad.df']

    el\_adj = arg\_dict['rend.unzoom']

    sc\_rays\_o, sc\_rays\_d, sc\_light\_dirs=[], [], []

    # On all training image angles

    for view\_i, img in enumerate(dataset['train\_imgs']):

        H, W, \_ = img.shape

        H\_sc = H//light\_df

        W\_sc = W//light\_df

        light\_angles = dataset['train\_light\_dirs'][view\_i]

        az, el = light\_angles[0,0], light\_angles[0,1]

        focal\_sc = dataset['train\_focals'][view\_i]/light\_df

        pose\_sc = data\_handling.pose\_spherical(az, -el, focal\_sc)

        if el\_adj:

            el\_view = dataset['train\_view\_dirs'][view\_i][0,1]

            rays\_o, rays\_d = get\_rays\_zoom(H\_sc, W\_sc, focal\_sc, pose\_sc, 1/np.sin(el\_view))

        else:

            rays\_o, rays\_d = get\_rays\_zoom(H\_sc, W\_sc, focal\_sc, pose\_sc, 1.0)

        rays\_o = tf.reshape(rays\_o, [H\_sc, W\_sc, 3])

        rays\_d = tf.reshape(rays\_d, [H\_sc, W\_sc, 3])

        sc\_rays\_o.append(rays\_o)

        sc\_rays\_d.append(rays\_d)

        sc\_light\_dirs.append(tf.ones([rays\_o.shape[0],rays\_o.shape[1],1])@light\_angles)

    sc\_rays\_o = tf.reshape(tf.convert\_to\_tensor(sc\_rays\_o), [-1,3])

    sc\_rays\_d = tf.reshape(tf.convert\_to\_tensor(sc\_rays\_d), [-1,3])

    sc\_light\_dirs=tf.reshape(tf.convert\_to\_tensor(sc\_light\_dirs), [-1,2])

    return {'rays\_o':sc\_rays\_o, 'rays\_d':sc\_rays\_d, 'light\_dirs':sc\_light\_dirs}

def generate\_custom\_light\_correction\_rays(dataset, arg\_dict):

    """

    Generate ray set for solar correction based on custom angles.

    Two modes are possible.

    1. 'linear' for angles equally spread between a start and end direction.

    2. 'rectangle' for angles on a rectangluar grid, using start and end as corners.

    Parameters:

    dataset (dict): dataset as specified in 'data\_handling.generate\_dataset'

    arg\_dict (dict): config variables

    Returns:

    all\_train(dict): set of custom solar\_correction rays.

    """

    zoom\_factor=1.0

    base\_i=0

    light\_df = arg\_dict['train.shad.df']

    mode=arg\_dict['train.shad.custom']

    bounds\_start = arg\_dict['train.shad.custom.bounds.start']

    bounds\_end = arg\_dict['train.shad.custom.bounds.end']

    # Convert to radian

    bounds = np.deg2rad(np.array([bounds\_start, bounds\_end]))

    n\_ray\_images = arg\_dict['train.shad.custom.bounds.samp']

    if mode == 'linear':

        # Linear interpolation from bounds[0] to bounds[1]

        n\_ray\_images = n\_ray\_images[0]

        azs = tf.cast(tf.linspace(bounds[0,0], bounds[1,0], n\_ray\_images), dtype=def\_dtype)

        els = tf.cast(tf.linspace(bounds[0,1], bounds[1,1], n\_ray\_images), dtype=def\_dtype)

    if mode == 'rectangle':

        # Sample regularly on the rectangle defined by bounds, following the sampling pattern given

        azs = tf.cast(tf.linspace(bounds[0,0], bounds[1,0], n\_ray\_images[0]), dtype=def\_dtype)

        els = tf.cast(tf.linspace(bounds[0,1], bounds[1,1], n\_ray\_images[1]), dtype=def\_dtype)

        n\_ray\_images = n\_ray\_images[0]\*n\_ray\_images[1]

        azs, els = tf.meshgrid(azs, els)

    azs = tf.reshape(azs, [-1])

    els = tf.reshape(els, [-1])

    H, W, \_ = dataset['train\_imgs'][base\_i].shape

    H\_sc = H//light\_df

    W\_sc = W//light\_df

    focal\_sc = dataset['train\_focals'][base\_i]/light\_df

    sc\_rays\_o, sc\_rays\_d, sc\_light\_dirs=[], [], []

    for i in range(n\_ray\_images):

        az, el = azs[i], els[i]

        sc\_view\_dir = tf.reshape(tf.convert\_to\_tensor([az, el], dtype=def\_dtype), [1,2])

        pose\_sc = data\_handling.pose\_spherical(az, -el, focal\_sc)

        rays\_o, rays\_d = get\_rays\_zoom(H\_sc, W\_sc, focal\_sc, pose\_sc, zoom\_factor)

        rays\_o = tf.reshape(rays\_o, [H\_sc, W\_sc, 3])

        rays\_d = tf.reshape(rays\_d, [H\_sc, W\_sc, 3])

#         rays\_o = array\_to\_patches(rays\_o, bc\_size)

        sc\_rays\_o.append(rays\_o)

        sc\_rays\_d.append(rays\_d)

        sc\_light\_dirs.append(tf.ones([rays\_o.shape[0],rays\_o.shape[1],1])@sc\_view\_dir)

    sc\_rays\_o = tf.reshape(tf.convert\_to\_tensor(sc\_rays\_o), [-1,3])

    sc\_rays\_d = tf.reshape(tf.convert\_to\_tensor(sc\_rays\_d), [-1,3])

    sc\_light\_dirs = tf.reshape(tf.convert\_to\_tensor(sc\_light\_dirs), [-1,2])

    return {'rays\_o':sc\_rays\_o, 'rays\_d':sc\_rays\_d, 'light\_dirs':sc\_light\_dirs}

def shuffle\_rays(rays):

    """

    Randomly shuffle rays using tf.random.shuffle.

    Parameters:

    rays (dict): set of rays to shuffle

    Returns:

    rays (dict): shuffled rays

    """

    rand\_indices = tf.random.shuffle(tf.convert\_to\_tensor(range(rays['rays\_o'].shape[0]), dtype=tf.int32))

    return {k: tf.gather(v, rand\_indices) for k, v in iter(rays.items())}

def get\_ray\_batch(rays, start, end):

    """Retrieve subset of rays from start index to end index"""

    sub\_rays = {k: v[start:end,...] for k, v in iter(rays.items())}

    return sub\_rays

def concat\_rays(rays1, rays2):

    """Group two sets of rays"""

    return {k: tf.concat([v, rays2[k]], axis=0) for k, v in iter(rays1.items())}

#### Sampling

def sample\_alt(rays\_o, rays\_d, bounds, N\_samples):

    """

    Sample along the rays, using the altitude bounds.

    Parameters:

    rays\_o (Tensor[H\*W, 3]): rays origin.

    rays\_d (Tensor[H\*W, 3]): rays direction.

    bounds (float, float): minimum and maximum altitude.

    N\_samples (int): number of samples along the rays.

    Returns:

    z\_vals (Tensor[H\*W, N\_samples]): depth values along the rays.

    """

    alt\_min, alt\_max = def\_dtype(bounds[0]), def\_dtype(bounds[1])

    alt\_vals = tf.linspace(alt\_max, alt\_min, N\_samples)

    z\_vals = (alt\_vals - rays\_o[...,None,2])/rays\_d[...,None,2]

    return z\_vals

def sample\_nf(bounds, N\_samples):

    """

    Sample using the near far distance.

    Parameters:

    bounds (float, float): near and far distance.

    N\_samples (int): number of samples.

    Returns:

    z\_vals (Tensor[N\_samples]): depth values.

    """

    near, far = bounds[0], bounds[1]

    z\_vals = tf.linspace(near, far, N\_samples)

    return z\_vals

def uniform\_bin\_sampling(z\_vals):

    """

    Randomly perturb a tensor of depth values of multiple rays.

    Each sample is pushed between its current position and the position of the next sample by a random amount, to achieve a continuous integration over a large number of iterations.

    Parameters:

    z\_vals (Tensor[N\_rays, N\_samples]): Initial depth values.

    Returns:

    z\_vals (Tensor[N\_rays, N\_samples]): Perturbed depth values.

    """

    # get intervals between samples

    mids = .5 \* (z\_vals[..., 1:] + z\_vals[..., :-1])

    upper = tf.concat([mids, z\_vals[..., -1:]], -1)

    lower = tf.concat([z\_vals[..., :1], mids], -1)

    # stratified samples in those intervals

    t\_rand = tf.random.uniform(z\_vals.shape)

    z\_vals = lower + (upper - lower) \* t\_rand

    return z\_vals

def sample\_pdf(bins, weights, N\_samples, det=False):

    """

    Sample new integration depths from a discrete probability density function given by bins and weights.

    Parameters:

    bins (Tensor[N\_b]): Depth values.

    weights (Tensor[N\_b]): Importance of each bin.

    N\_samples (int): number of samples along the ray.

    det (bool) : random sampling in the pdf

    Returns:

    samples (Tensor[N\_samples]): New samples.

    """

    # Get pdf

    weights += 1e-5  # prevent nans

    pdf = weights / tf.reduce\_sum(weights, -1, keepdims=True)

    cdf = tf.cumsum(pdf, -1)

    cdf = tf.concat([tf.zeros\_like(cdf[..., :1]), cdf], -1)

    # Take uniform samples

    if det:

        u = tf.linspace(0., 1., N\_samples)

        u = tf.broadcast\_to(u, list(cdf.shape[:-1]) + [N\_samples])

    else:

        u = tf.random.uniform(list(cdf.shape[:-1]) + [N\_samples])

    # Invert CDF

    inds = tf.searchsorted(cdf, u, side='right')

    below = tf.maximum(0, inds-1)

    above = tf.minimum(cdf.shape[-1]-1, inds)

    inds\_g = tf.stack([below, above], -1)

    cdf\_g = tf.gather(cdf, inds\_g, axis=-1, batch\_dims=len(inds\_g.shape)-2)

    bins\_g = tf.gather(bins, inds\_g, axis=-1, batch\_dims=len(inds\_g.shape)-2)

    denom = (cdf\_g[..., 1]-cdf\_g[..., 0])

    denom = tf.where(denom < 1e-5, tf.ones\_like(denom), denom)

    t = (u-cdf\_g[..., 0])/denom

    samples = bins\_g[..., 0] + t \* (bins\_g[..., 1]-bins\_g[..., 0])

    return samples

def resample\_importance(z\_vals, weights, N\_importance, render\_mode='nf'):

    """

    Importance sampling.

    Obtain additional integration samples to evaluate based on the alpha-compositing weights.

    Parameters:

    z\_vals (Tensor[N\_samples]): Initial sample depths.

    weights (Tensor[N\_samples]): Alpha-compositing weights.

    N\_importance (int): Number of importance samples along the ray

    Returns:

    z\_vals (Tensor[N\_samples+N\_importance]): all depth values (initial + importance).

    """

    z\_vals\_mid = .5 \* (z\_vals[..., 1:] + z\_vals[..., :-1])

    if render\_mode == 'nf':

        # Sample according to the probability density function

        z\_samples = sample\_pdf(z\_vals\_mid, weights[...,1:-1], N\_importance, True)

    elif render\_mode == 'alt':

        # Add first and last points to fix edge cases where weights is 1 only on the first or last element

        # Not the case for near-far rendering where the final distance is "infinity"

        z\_vals\_mid = tf.concat([z\_vals[...,0:1], z\_vals\_mid, z\_vals[...,-1:]], axis=-1)

        # Sample according to the probability density function

        z\_samples = sample\_pdf(z\_vals\_mid, weights, N\_importance, True)

    else:

        print("Unrecognized rendering mode")

        return

    z\_samples = tf.stop\_gradient(z\_samples)

    # Obtain all points to evaluate color, density at.

    z\_vals = tf.sort(tf.concat([z\_vals, z\_samples], -1), -1)

    return z\_vals

#### Rendering

def render\_rays(model, arg\_dict, ray\_batch, rets=['rgb'], rand=False, raw\_noise\_std=(0.0, 0.0), rescale\_factor=None, chunk=1024\*256):

    """

    Main rendering function.

    Render outputs of a set of rays, used both for training and for inference.

    Parameters:

    model (dict): TF model, dimensions and embedding functions as defined in models.generate\_model

    arg\_dict (dict): Global configuration variables

    ray\_batch (dict): Target rays for rendering

    rets (list(string)): rendering outputs (see pts2outputs)

    rand (bool): Randomly perturb sample positions. Set to True for training and False for inference.

    raw\_noise\_std (float, float): Resp. strength of noise on opacity and on shadow outputs.

    rescale\_factor (float): Global rescale factor for dataset.

    chunk (int): Chunking parameter for memory management, reduce if overflow (slower execution).

    Returns:

    ret\_dict (dict): desired outputs according to 'rets' parameter

    """

    N\_samples = arg\_dict["rend.nsamples"]

    N\_importance = arg\_dict["rend.nimportance"]

    fine\_render = (N\_importance > 0)

    if rescale\_factor is None:

        rescale\_factor = arg\_dict["rend.rescale"]

    # True sampling distance is original sampling distance multiplied by added image downsample factor

    image\_sd = arg\_dict['data.image.sd'] \* arg\_dict['data.image.df']

    # Convert bounds from original units to pixel units

    if arg\_dict['rend.mode'] == 'alt':

        bounds = [arg\_dict['rend.mode.alt.min']/image\_sd, arg\_dict['rend.mode.alt.max']/image\_sd]

    elif arg\_dict['rend.mode'] == 'nf':

        bounds = [arg\_dict['rend.mode.nf.near']/image\_sd, arg\_dict['rend.mode.nf.far']/image\_sd]

    view\_dirs = ray\_batch['view\_dirs'] if 'view\_dirs' in ray\_batch.keys() else None

    light\_dirs = ray\_batch['light\_dirs'] if 'light\_dirs' in ray\_batch.keys() else None

    # Sample z values (distance along ray from ray\_o)

    if arg\_dict["rend.mode"] == 'alt':

        z\_vals = sample\_alt(ray\_batch['rays\_o'], ray\_batch['rays\_d'], bounds, N\_samples)

    elif arg\_dict["rend.mode"] == 'nf':

        z\_vals = sample\_nf(bounds, N\_samples)

    else:

        print("Unrecognized rendering mode " + arg\_dict["rend.mode"])

        return

    # Randomly perturb z values

    if rand:

        z\_vals = uniform\_bin\_sampling(z\_vals)

    # Calculate x,y,z position of query points in absolute ref frame

    pts = ray\_batch['rays\_o'][...,None,:] + ray\_batch['rays\_d'][...,None,:] \* z\_vals[...,:,None]

    pts\_flat = tf.reshape(pts, [-1,3])

    # Perform the rendering

    if fine\_render:

        # First pass : coarse rendering to estimate weights

        ret\_dict\_coarse = pts2outputs(model, pts\_flat, z\_vals, N\_samples, light\_dirs, view\_dirs, norm=rescale\_factor, raw\_noise\_std=raw\_noise\_std, mode=arg\_dict["rend.mode"], rets=['weights', 'z\_vals'], chunk=chunk)

        # Obtain new sample positions

        z\_vals = resample\_importance(ret\_dict\_coarse['z\_vals'], ret\_dict\_coarse['weights'], N\_importance, render\_mode=arg\_dict["rend.mode"])

        pts = ray\_batch['rays\_o'][..., None, :] + ray\_batch['rays\_d'][..., None, :] \* z\_vals[..., :, None]  # [N\_rays, N\_samples + N\_importance, 3]

        pts\_flat = tf.reshape(pts, [-1,3])

        # Final pass on all samples

    ret\_dict = pts2outputs(model, pts\_flat, z\_vals, N\_samples + N\_importance, light\_dirs, view\_dirs,  norm=rescale\_factor, raw\_noise\_std=raw\_noise\_std, mode=arg\_dict["rend.mode"], rets=rets, chunk=chunk)

    return ret\_dict

def pts2raw(pts\_flat, network\_fn, embed\_fn\_pos):

    """

    Convert points to raw (unactivated) network outputs. These correspond to the density and 3D color (radiance), and optionally solar visiblity and sky color.

    Parameters:

    pts\_flat (Tensor[N\_pts, 3 + L\_embed\_dir]): 3D Points to apply the network.

    network\_fn (keras.model): Function of the neural network.

    embed\_fn\_pos (function): positional embedding function as defined in models.py

    Returns:

    raw (Tensor[N\_pts, N\_outputs]: model outputs

    """

    # Positional encoding

    if embed\_fn\_pos is not None:

        pts\_flat = tf.concat([embed\_fn\_pos(pts\_flat[:,:3]), pts\_flat[:,3:]], axis=-1)

    # Apply network function

    raw = network\_fn(pts\_flat)

    return raw

def dir\_encoding(pts\_flat, view\_dirs, light\_dirs, N\_samples, embed\_fn\_dir):

    """

    Add directional encoding to a set of points.

    Parameters:

    pts\_flat (Tensor[N\_rays\*N\_samples, 3]): 3D Points to apply the encoding.

    view\_dirs (Tensor[N\_rays, 2]): Viewing directions.

    light\_dirs (Tensor[N\_rays, 2]): Lighting directions.

    N\_samples (int): Number of samples along each ray.

    embed\_fn\_dir (function): directional embedding function as defined in models.py

    Returns:

    pts\_flat (Tensor[N\_rays\*N\_samples, 3 + L\_embed\_dir]): Points with directional embedding.

    """

    if view\_dirs is not None:

        view\_dirs = tf.broadcast\_to(view\_dirs[...,None], [view\_dirs.shape[0], view\_dirs.shape[1] , N\_samples])

        view\_dirs = tf.transpose(view\_dirs, perm=[0,2,1])

        v\_dirs\_flat = tf.reshape(view\_dirs, [-1,2])

        v\_dirs\_flat = embed\_fn\_dir(v\_dirs\_flat)

        pts\_flat = tf.concat([pts\_flat, v\_dirs\_flat], axis=-1)

    if light\_dirs is not None:

        light\_dirs = tf.broadcast\_to(light\_dirs[...,None], [light\_dirs.shape[0], light\_dirs.shape[1] , N\_samples])

        light\_dirs = tf.transpose(light\_dirs, perm=[0,2,1])

        s\_dirs\_flat = tf.reshape(light\_dirs, [-1,2])

        s\_dirs\_flat = embed\_fn\_dir(s\_dirs\_flat)

        pts\_flat = tf.concat([pts\_flat, s\_dirs\_flat], axis=-1)

    return pts\_flat

def batchify(fn, chunk=1024\*256):

    """

    Batching function to reduce memory footprint during ray rendering.

    Parameters:

    fn (function): Function to be applied to inputs.

    Returns:

    fn\_batch (function): Function that will apply fn to inputs in batches.

    """

    return lambda inputs : tf.concat([fn(inputs[i:i+chunk,:]) for i in range(0, inputs.shape[0], chunk)], 0)

def pts2outputs(model, pts\_flat, z\_vals, N\_samples, light\_dirs=None, view\_dirs=None, norm=1.0, raw\_noise\_std=(1.0, 1.0), mode='nf', rets=['rgb'], chunk=1024\*256):

    """

    Compute outputs of a set of points.

    Performs alpha-compositing for a set of target points. Runs the network inference and shading model. Outputs are only generated if requested in the rets parameter.

    Parameters:

    model (dict): TF model, dimensions and embedding functions as defined in models.generate\_model

    pts\_flat (Tensor[N\_rays\*N\_samples, 3]): Points to compute outputs of alpha-compositing

    z\_vals (Tensor[N\_rays\*N\_samples]): Target depths for rendering

    N\_samples (int): Number of samples along each ray.

    view\_dirs (Tensor[N\_rays, 2]): Viewing directions.

    light\_dirs (Tensor[N\_rays, 2]): Lighting directions.

    norm (float): Rescale factor given by calculate\_rescale\_factor

    raw\_noise\_std (float, float): Resp. strength of noise on opacity and on shadow outputs.

    mode (string): 'nf' or 'alt\_max' depending on the sampling mode.

    rets (list(string)): Requested rendering outputs.

    chunk (int): Chunking parameter for memory management, reduce if overflow (slower execution).

    Returns:

    ret\_dict (dict): Dictionary of outputs with the following key-value associations.

    rgb (Tensor[N\_rays, 3]): integrated (and shaded) RGB

    depth (Tensor[N\_rays]): estimated surface depth

    weights (Tensor[N\_rays, N\_samples]): alpha-compositing weights

    trans (Tensor[N\_rays]): accumulated transparency (final trans value)

    acc (Tensor[N\_rays]): sum of alpha-compositng weights

    z\_vals (Tensor[N\_rays\*N\_samples]): depth values used for rendering

    no\_shadow (Tensor[N\_rays, 3]): albedo rendering (no shadows)

    sky\_only (Tensor[N\_rays, 3]): only sky illumination (no direct light)

    ret\_sun (Tensor[N\_rays]): integrated solar visibility function

    ret\_shadow\_loss (Tensor[N\_rays]): Shadow loss between solar visibility and transparency along solar ray

    sky (Tensor[N\_rays, 3]): visualize irradiance at surface

    """

    # Normalize so that the largest extent fits within a -1, 1 cube

    if norm != 1.0:

        pts\_flat = pts\_flat/norm

    # Extract model parameters

    network\_fn = model["model"]

    embed\_fns = model["emb"]

    model\_dims = model["dim"]

    n\_in = np.sum(np.array(model\_dims["in"]))

    n\_out = np.sum(np.array(model\_dims["out"]))

    pts\_flat = dir\_encoding(pts\_flat, view\_dirs, light\_dirs, N\_samples, embed\_fns[1])

    # Convert points to raw values with batching for memory

    raw = batchify(lambda pts: pts2raw(pts, network\_fn, embed\_fns[0]))(pts\_flat)

    raw = tf.reshape(raw, [-1 , N\_samples, n\_out])

    # Add noise to the opacity prediciton to regularlize

    noise, noise\_s = 0., 0.

    if raw\_noise\_std[0] > 0:

        noise = tf.random.normal(raw[..., 3].shape) \* raw\_noise\_std[0]

    if (raw\_noise\_std[1] > 0) and (model\_dims["out"][2] != 0):

        noise\_s = tf.random.normal(raw[..., 4].shape) \* raw\_noise\_std[1]

    # Compute opacities and colors

    sigma\_a = tf.nn.relu(raw[...,3] + noise)

    rgb = tf.math.sigmoid(raw[...,:3])

    # Do volume rendering

    # Rescale distances w.r.t rescale factor

    dists = (z\_vals[..., 1:] - z\_vals[..., :-1])/norm

    if mode == 'alt':

        #Replicate last distance as distance for last point

        dists = tf.concat([dists, dists[...,-2:-1]], axis = -1)

        alpha = 1.-tf.exp(-sigma\_a \* dists)

        trans = tf.math.cumprod(1.-alpha + 1e-10, -1, exclusive=True)

        weights = alpha \* trans

        #Replace last weight with sum of others to always get sum of weights = 1

        last\_weight = tf.convert\_to\_tensor(1.0-tf.reduce\_sum(weights[...,:-1], axis=-1))

        last\_weight = tf.reshape(last\_weight, [-1,1])

        weights = tf.concat([weights[...,:-1], last\_weight], axis=-1)

    elif mode == 'nf':

        # The 'distance' from the last integration time is infinity

        dists = tf.concat([dists, tf.broadcast\_to([1e10], dists[..., :1].shape)], axis = -1)

        alpha = 1.-tf.exp(-sigma\_a \* dists)

        trans = tf.math.cumprod(1.-alpha + 1e-10, -1, exclusive=True)

        weights = alpha \* trans

    # Shading options

    if (model\_dims["out"][2] == 0) and (model\_dims["out"][3] == 0):

        #No shading

        integrated\_rgb = tf.reduce\_sum(weights[...,None] \* rgb, -2)

    if (model\_dims["out"][2] == 1) and (model\_dims["out"][3] == 0):

        #Sun light only

        s = tf.nn.sigmoid(raw[...,4] + noise\_s)

        integrated\_rgb = tf.reduce\_sum(weights[...,None] \* rgb \* s, -2)

    if (model\_dims["out"][2] == 0) and (model\_dims["out"][3] == 3):

        #Sky light only

        sky = tf.math.sigmoid(raw[...,4:7])

        integrated\_rgb = tf.reduce\_sum(weights[...,None] \* rgb \* sky, -2)

    if (model\_dims["out"][2] == 1) and (model\_dims["out"][3] == 3):

        #Sun + Sky

        s = tf.nn.sigmoid(raw[...,4] + noise\_s)

        sky = tf.math.sigmoid(raw[...,5:8])

        li = s[...,None] + sky\*(1.0-s[...,None])

        integrated\_rgb = tf.reduce\_sum(weights[...,None] \* rgb \* li, -2)

    # Return dictionary

    ret\_dict={}

    if 'rgb' in rets:

        ret\_dict['rgb'] = integrated\_rgb

    if 'depth' in rets:

        ret\_dict['depth'] = tf.reduce\_sum(weights \* z\_vals, -1)

    if 'weights' in rets:

        ret\_dict['weights'] = weights

    if 'trans' in rets:

        ret\_dict['trans'] = trans[:, -1]

    if 'acc' in rets:

        ret\_dict['acc'] = tf.reduce\_sum(weights, -1)

    if 'z\_vals' in rets:

        ret\_dict['z\_vals'] = z\_vals

    if 'no\_shadow' in rets:

        ret\_dict['no\_shadow'] = tf.reduce\_sum(weights[...,None] \* rgb, -2)

    if 'sky\_only' in rets:

        ret\_dict['sky\_only'] = tf.reduce\_sum(weights[...,None] \* rgb \* sky, -2)

    if 'ret\_sun' in rets:

        ret\_dict['ret\_sun'] = tf.reduce\_sum(tf.stop\_gradient(weights[...,None]) \* s[...,np.newaxis], -2)

    if 'ret\_shadow\_loss' in rets:

        ret\_dict['ret\_shadow\_loss']= tf.reduce\_mean(tf.math.square(tf.stop\_gradient(trans[..., None] + weights[...,None]) - s[...,None]), -2)

    if 'sky' in rets:

        ret\_dict['sky'] = tf.reduce\_sum(weights[...,None]\*(s[...,None] + sky\*(1.0-s[...,None])),-2)

    return ret\_dict

def render\_image(model, arg\_dict, hwf, pose, zoom\_factor, light\_dirs, view\_dirs, rets=['rgb']):

    """

    Compute an output image for one pose configuration.

    Similar to render\_rays but for an image, the outputs are defined in the same way.

    Parameters:

    model (dict): TF model, dimensions and embedding functions as defined in models.generate\_model

    arg\_dict (dict): Global configuration variables

    hwf (int, int, float): H, W and Focal of requested image.

    zoom\_factor: Optional zoom factor

    light\_dirs (Tensor[1, 2]): Lighting direction.

    view\_dirs (Tensor[1, 2]): Viewing direction.

    rets (list(string)): Requested rendering outputs.

    Returns:

    ret\_dict (dict): Dictionary of outputs reshaped to [H, W, x] where x can be 3 or 1 depending on which outputs is requested.

    """

    H, W, focal = hwf

    el = view\_dirs[0,1]

    ray\_batch = {}

    # Handle special unzoom mode for satellite images

    if arg\_dict['rend.unzoom']:

        ray\_batch['rays\_o'], ray\_batch['rays\_d'] = get\_rays\_zoom(H, W, focal, pose, zoom\_factor/np.sin(el))

    else:

        ray\_batch['rays\_o'], ray\_batch['rays\_d'] = get\_rays\_zoom(H, W, focal, pose, zoom\_factor)

    # Constant view and light directions across the scene

    if arg\_dict['model.ins.views']:

        ray\_batch['view\_dirs'] = tf.ones([H\*W,1])@view\_dirs

    if arg\_dict['model.ins.light']:

        ray\_batch['light\_dirs'] = tf.ones([H\*W,1])@light\_dirs

        ray\_batch['light\_vectors'] = tf.broadcast\_to(data\_handling.pose\_spherical(light\_dirs[0, 0], -light\_dirs[0, 1], 1.0)[:3,2], [H\*W,3])

    # Rescale factor for image size H, W

    diag = semi\_diagonal(H, W)

    # Ray rendering

    ret\_dict = render\_rays(model, arg\_dict, ray\_batch, rets=rets, rand=False, raw\_noise\_std=(0.0, 0.0), rescale\_factor=diag, chunk=1024\*128)

    ## Reshape results to [H, W, ...]

    if 'rgb' in rets:

        ret\_dict['rgb'] = tf.reshape(tf.clip\_by\_value(ret\_dict['rgb'], 0.0, 1.0) ,[H, W, 3])

    if 'depth' in rets:

        ret\_dict['depth'] = tf.reshape(ret\_dict['depth'], [H, W])

    if 'acc' in rets:

        ret\_dict['acc'] = tf.reshape(ret\_dict['acc'], [H, W])

    if 'no\_shadow' in rets:

        ret\_dict['no\_shadow'] = tf.reshape(tf.clip\_by\_value(ret\_dict['no\_shadow'], 0.0, 1.0), [H, W, 3])

    if 'sky\_only' in rets:

        ret\_dict['sky\_only'] = tf.reshape(ret\_dict['sky\_only'], [H, W, 3])

    if 'ret\_sun' in rets:

        ret\_dict['ret\_sun'] = tf.reshape(ret\_dict['ret\_sun'],[H, W])

    if 'ret\_shadow\_loss' in rets:

        ret\_dict['ret\_shadow\_loss'] = tf.reshape(ret\_dict['ret\_shadow\_loss'],[H, W])

    if 'sky' in rets:

        ret\_dict['sky'] =  tf.reshape(tf.clip\_by\_value(ret\_dict['sky'], 0.0, 1.0),[H, W, 3])

    return ret\_dict

def render\_dataset(dataset, model, rets, arg\_dict, zoom\_factor=1.0):

    """

    Compute outputs for the whole dataset.

    Similar to render\_image but for a dataset, the outputs are defined in the same way.

    Parameters:

    dataset (dict):

    model (dict): TF model, dimensions and embedding functions as defined in models.generate\_model

    rets (list(string)): Requested rendering outputs.

    arg\_dict (dict): Global configuration variables

    zoom\_factor: Optional zoom factor

    Returns:

    rendered\_images (dict): Two separate lists of train and test rendering dicts (see render\_image).

    """

    rendered\_images={'train\_rend':[], 'test\_rend':[]}

    # Train images

    train\_imgs = dataset['train\_imgs']

    poses, focals, view\_dirs, light\_dirs = dataset['train\_poses'], dataset['train\_focals'], dataset['train\_view\_dirs'], dataset['train\_light\_dirs']

    for img, pose, focal, view\_dir, light\_dir in zip(train\_imgs, poses, focals, view\_dirs, light\_dirs):

        hwf = img.shape[0], img.shape[1], focal

        ret\_dict = render\_image(model, arg\_dict, hwf, pose, zoom\_factor, light\_dir, view\_dir, rets=rets)

        rendered\_images['train\_rend'].append(ret\_dict)

    # Test images

    test\_imgs = dataset['test\_imgs']

    poses, focals, view\_dirs, light\_dirs = dataset['test\_poses'], dataset['test\_focals'], dataset['test\_view\_dirs'], dataset['test\_light\_dirs']

    for img, pose, focal, view\_dir, light\_dir in zip(test\_imgs, poses, focals, view\_dirs, light\_dirs):

        hwf = img.shape[0], img.shape[1], focal

        ret\_dict = render\_image(model, arg\_dict, hwf, pose, zoom\_factor, light\_dir, view\_dir, rets=rets)

        rendered\_images['test\_rend'].append(ret\_dict)

    return rendered\_images

def semi\_diagonal(H, W):

    """Semi-diagonal of a rectangle"""

    return np.sqrt((H/2)\*\*2 + (W/2)\*\*2)

def calculate\_rescale\_factor(dataset):

    """Compute the rescale factor, used to ensure that the scene stays within the [-1, 1] bounds for the network."""

    H, W, \_ = dataset['train\_imgs'][0].shape

    return semi\_diagonal(H, W)