Coded Image Generation: [30pts]

The goal of this project is to obtain a simulated coded (or 'blurred') image from a given RGB image, Ground Truth Metric Depth Image and Point Spread Function of the phase mask.

Input RGB Image:

Input Metric Depth Image:

Output Coded RGB Image:

Mount Your Google Drive and Download PSF file from CodedVO Github:

```
# Step 1: Mount Google Drive to access and save files directly to your
drive
# from google.colab import drive

# # Mount Google Drive
# drive.mount('/content/drive')

# Download and Save 'phasecam-psf-27.npy' from
https://github.com/naitri/CodedVO/blob/main/phasecam-psfs-27.npy

# Link to the datasets: (Use only RGB and Depth Images)
#
https://drive.google.com/drive/folders/12GrDxTBMaSlGeMRWycxmCQl01BHnC5
-O
```

Import Environments:

```
import os          # Importing the os module for interacting with the
operating system
import cv2         # Importing OpenCV, a computer vision library
import numpy as np # Importing numpy for numerical operations
import torch       # Importing PyTorch for tensor operations and GPU
acceleration
import tqdm        # Importing tqdm for creating progress bars
import argparse    # Importing argparse for command-line argument
```

```python
parsing
os.environ["OPENCV_IO_ENABLE_OPENEXR"] = "1"
import sys
# Enable support for reading and writing OpenEXR files in OpenCV


# Set the device to 'cuda' (GPU) if available, otherwise use 'cpu'
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)
# Enable cuDNN support for optimized GPU computation
torch.backends.cudnn.enabled = True

# Enable cuDNN benchmarking, which allows PyTorch to optimize
performance for the first time it's run
torch.backends.cudnn.benchmark = True
```

```
cuda
```

PyTorch Tensor to Numpy Conversion

```python
def tensor_to_numpy_image(tensor: torch.Tensor) -> np.ndarray:
    """
    Convert a PyTorch tensor to a NumPy image.

    Args:
        tensor (torch.Tensor): Input tensor with shape (C, H, W),
                               where C is the number of channels,
                               H is the height, and W is the width.

    Returns:
        np.ndarray: Output image in NumPy format (H, W, C) with pixel
values
                    in the range [0, 255].
    """

    # Move the channel axis from the first dimension (C, H, W) to the
last dimension (H, W, C)
    out = tensor.moveaxis(0, -1)

    # Clamp the values to the range [0, 255], cast to uint8 type for
image format,
    # move the tensor to CPU if it's on GPU, and convert to NumPy
array.
    return torch.clamp(out, 0, 255).to(torch.uint8).cpu().numpy()
```

Get Depth Layers

Get depth layers from a depth map by quantizing depth values into discrete layers.

```
def get_depth_layers(depth_map: torch.Tensor, depth_layers:
torch.Tensor) -> torch.Tensor:
    """
    Get depth layers from a depth map by quantizing depth values into
discrete layers.

    Args:
        depth_map (torch.Tensor): Input depth map, a 2D or 3D tensor
where each value represents
                                  the depth (distance) at a specific
pixel.
        depth_layers (torch.Tensor): Defined depth layers, a 1D tensor
that contains threshold values
                                     to separate the depth map into
discrete layers.

    Returns:
        torch.Tensor: A tensor where each layer corresponds to a
binary mask indicating
                      whether each pixel belongs to a specific depth
layer.
    """

    # Move the depth_layers tensor to the same device as depth_map for
efficient computation.
    # This ensures that both tensors are compatible for GPU or CPU
processing.
    depth_layers = depth_layers.to(depth_map.device)

    # torch.bucketize is a function that quantizes the depth_map
values based on depth_layers.
    # Mathematically, it assigns each pixel in the depth_map to a
"bin" (depth layer).
    # The input depth_map is a tensor of real values, and the
depth_layers tensor contains discrete
    # boundary values (think of it as intervals that represent
different ranges of depth).
    # For example:
    # If depth_layers = [0.5, 1.0, 1.5] and the depth_map contains
values like [0.3, 0.6, 1.2, 1.8],
    # torch.bucketize assigns a bin index to each depth value:
    # 0.3 -> bin 0 (less than 0.5),
    # 0.6 -> bin 1 (between 0.5 and 1.0),
    # 1.2 -> bin 2 (between 1.0 and 1.5),
    # 1.8 -> bin 3 (greater than 1.5).
```

```
######################################################################
################################################
    # WRITE A FUNCTION TO OBTAIN QUANTIZED DEPTH FROM 'depth_map' AND
'depth_layers' USING 'torch.bucketize' FUNCTION

######################################################################
################################################
    quantized_depth = torch.bucketize(depth_map, depth_layers)
    # print("Quantized Depth",np.shape(quantized_depth))

    # torch.stack and list comprehension:
    # For each depth layer, create a binary mask where the value is 1
if the pixel belongs to that layer
    # and 0 otherwise. The quantized_depth == j comparison creates a
binary mask for each j in the
    # range of the number of depth layers.

    # Explanation of range(len(depth_layers)):
    # len(depth_layers) returns the number of depth intervals.
    # For each interval, we compare the quantized depth values with
the corresponding layer index (j).
    # This results in a set of binary masks, each representing a
specific depth layer.

    # Example:
    # If quantized_depth = [0, 1, 2, 3] and len(depth_layers) = 4,
this will return four binary masks:
    # Mask 0: True where quantized_depth == 0 (first depth layer),
False elsewhere.
    # Mask 1: True where quantized_depth == 1 (second depth layer),
False elsewhere.
    # Mask 2: True where quantized_depth == 2 (third depth layer),
False elsewhere.
    # Mask 3: True where quantized_depth == 3 (fourth depth layer),
False elsewhere.


######################################################################
################################################
    # WRITE A FUNCTION TO OBTAIN DEPTH LAYERS STACK FROM
'quantized_depth' AND 'depth_layers' USING 'torch.stack' FUNCTION

######################################################################
################################################
    depth_layers_stack = torch.stack([(quantized_depth == i) for i in
range(len(depth_layers))], dim=0)
    # print(depth_layers_stack)
```

```python
    return depth_layers_stack
```

Single PSF Convolution Operation:

Perform convolution of the input image with a single Point Spread Function (PSF).

PSF (Point Spread Function): This describes how a point source of light is blurred or spread by an imaging system. Each depth and channel can have a unique PSF that models this blur.

```python
def single_psf_convolution(image: torch.Tensor, psfs: torch.Tensor,
depth_idx: int, channel_idx: int, padding: int) -> torch.Tensor:
    """
    Convolve image with a single PSF.

Args:
        image (torch.Tensor): Input image tensor, typically of shape
(N, C, H, W) where:
                              N - batch size, C - number of channels,
H - height, W - width.
        psfs (torch.Tensor): Tensor of Point Spread Functions (PSFs),
typically with shape (D, C, H, W) where:
                             D - depth layers, C - number of channels,
H - height, W - width.
        depth_idx (int): Index specifying which depth layer PSF to use
for convolution.
        channel_idx (int): Index specifying which channel PSF to use
for convolution.
        padding (int): Padding size to be applied during the
convolution to control output size.

    Returns:
        torch.Tensor: The result of convolving the image with the
selected PSF, output is of shape (N, C, H, W).
    """

    psfs = psfs.to(image.device).to(image.dtype)  # Ensures psfs is on
the GPU with the same dtype as the image
    # Write a function (Use torch's conv2d operator) that performs a
2D convolution operation between the input image and the PSF.
    psf_conv = torch.nn.functional.Conv2d(image,psfs) # or we can use
convolve_images()
    return psf_conv
```

Linear Convolution

The goal of this function is to apply different Point Spread Functions (PSFs) to the image based on the depth at each pixel. This allows for depth-dependent blur effects, where the blur at each pixel is determined by its corresponding depth value.

The linear model is given by: (For this model, the input RGBD image is quantized into K depth layers l_k, with $k=0$ being the furthest layer.)

$$b(\lambda)=\sum_{k=0}^{K-1} \mathrm{PSF}_k(\lambda)*l_k(\lambda)+\eta$$

Follow: https://www.computationalimaging.org/publications/deepopticsdfd/ [Equation 4] for notations.

```
def linear_convolution(image: torch.Tensor, depth_map: torch.Tensor, 
psfs: torch.Tensor, depth_layers: torch.Tensor, padding: int) -> 
torch.Tensor:
    """
    Perform depth-dependent convolution using a set of Point Spread 
Functions (PSFs), where each PSF is associated
    with a specific depth layer, and the final result is a depth-aware 
convolution of the image.

    Args:
        image (torch.Tensor): Input image tensor, typically of shape 
(C, H, W), where:
                              C - number of channels (e.g., 3 for 
RGB), H - height, W - width.
        depth_map (torch.Tensor): Depth map tensor of shape (H, W), 
where each value represents the depth of a pixel.
        psfs (torch.Tensor): Point Spread Functions (PSFs) tensor of 
shape (D, C, H, W), where:
                             D - number of depth layers, C - number of 
channels, H and W - height and width of each PSF.
        depth_layers (torch.Tensor): A tensor defining the discrete 
depth layers to segment the depth map.
        padding (int): Padding size for the convolution.

    Returns:
        torch.Tensor: The final convolved image after applying depth-
aware convolutions.
    """

    # Move the psfs tensor to the same device as the input image for 
efficient computation,
    # ensuring that both tensors are compatible for GPU or CPU 
processing.
    psfs = psfs.to(image.device)
```

```
    # Generate binary masks (depth layers) from the depth map using
the predefined depth_layers.
    # The result is a set of binary masks (depth_mask), each
indicating where pixels belong to a specific depth layer.
    depth_mask = get_depth_layers(depth_map, depth_layers)
    # print("Depth Mask",np.shape(depth_mask))
    # Perform convolution for each channel (e.g., Red, Green, Blue in
RGB images).
    # The final output will be a tensor of convolved images, stacked
across channels.
    C,H,W  = image.shape
    convolved_image = torch.zeros_like(image)


######################################################################
######
    # Write a function that performs a depth-dependent convolution on
each color
    # channel of an image using point spread functions (PSFs)
corresponding to
    # different depth layers, sums the results across depth layers,
and then
    # stacks the convolved outputs for all channels into a final image
tensor.
    # Hint: Use torch.stack, torch.sum and conv2d functions

######################################################################
######
    for i in range(len(depth_layers)):
      mask = depth_mask[i] # retrieving the current depth layer
      masked_image = image*mask.unsqueeze(0)
      # print("MaskedImage ",np.shape(masked_image))# according to
pytorch site, not sure what it does
      convolved_channel = []
      for c in range(C):
        psf = psfs[i, c].unsqueeze(0).unsqueeze(0)  # Shape (1, 1,
H_psf, W_psf) for conv2d

        # Convolve masked image for each channel
        convolved = torch.nn.functional.conv2d(
            masked_image[c].unsqueeze(0).unsqueeze(0),  # Shape (1, 1,
H, W) for conv2d
            psf,  # Shape (1, 1, H_psf, W_psf)
            padding=padding
        )

      # convolved_layer = torch.stack(convolved_channel,dim=0)
      # convolved_layer_1 = convolved_layer[1:len(convolved_layer)]
      # print(np.shape(convolved_layer_1))
```

```
      # print(np.shape(convolved_image))
        convolved_image[c] += convolved.squeeze(0).squeeze(0)

    final_convolved_image = convolved_image
    return final_convolved_image
```

Non Linear Convolution: (Optional)

A linear model can accurately reproduce defocus blur for image regions corresponding to a locally constant depth value. However, this approach is incapable of accurately modeling defocus blur at depth discontinuities.

Thus, we will adopt a nonlinear differentiable image formation model based on alpha compositing and combine it with our wavelength and depth-dependent PSF as:

$$b(\lambda) = \sum_{k=0}^{K-1} \tilde{l}_k \prod_{k'=k+1}^{K-1} (1 - \tilde{\alpha}_{k'}) + \eta$$

Follow: https://www.computationalimaging.org/publications/deepopticsdfd/ [Equation 5] for notations.

```
def nonlinear_convolution():
  # Non linear code goes here...

  return final_convolved_image
```

Capture and Process all the image: (Nothing to write here)

Process all images and their corresponding depth maps in a folder using depth-dependent convolution.

```
def capture_image(img: np.ndarray, metric_depth: np.ndarray, psfs: 
torch.Tensor, depth_layers: torch.Tensor, padding: int) -> np.ndarray:
    """
    Capture and process an image using depth-dependent convolution.

    Args:
        img (np.ndarray): Input RGB image as a NumPy array, with shape 
(H, W, 3).
        metric_depth (np.ndarray): Metric depth map as a NumPy array, 
with shape (H, W), where each value represents the depth at that 
pixel.
        psfs (torch.Tensor): Tensor of Point Spread Functions (PSFs) 
of shape (D, C, H, W), used for depth-dependent blurring.
        depth_layers (torch.Tensor): Tensor defining the depth layers 
that segment the depth map.
        padding (int): Padding size to use during the convolution
```

```
operation.

    Returns:
        np.ndarray: The final processed image after applying depth-
dependent convolution, converted back to NumPy format.
    """

    # Convert the input RGB image from a NumPy array to a PyTorch
tensor.
    # `moveaxis(-1, 0)` changes the image from shape (H, W, 3) to (3,
H, W) to match PyTorch's channel-first convention.
    image = torch.from_numpy(img).moveaxis(-1,
0).to(torch.float32).to(device)

    # Convert the metric depth map from a NumPy array to a PyTorch
tensor, and move it to the appropriate device (GPU or CPU).
    depth = torch.from_numpy(metric_depth).to(device)

    # Perform linear depth-dependent convolution on the input image
using the given depth map, PSFs, and depth layers.
    coded = linear_convolution(image, depth, psfs, depth_layers,
padding)

    # Convert the resulting convolved image (PyTorch tensor) back to a
NumPy array in RGB format (H, W, 3).
    return tensor_to_numpy_image(coded)

def process_folder(root: str, psfs: torch.Tensor, depth_layers:
torch.Tensor, use_nonlinear: bool, is_blender: bool = False,
scale_factor: float = 5000):
    """
    Process all images and their corresponding depth maps in a folder
using depth-dependent convolution.

    Args:
        root (str): Root directory containing "rgb" (image) and
"depth" folders.
        psfs (torch.Tensor): Tensor of Point Spread Functions (PSFs)
to be applied for depth-dependent blurring.
        depth_layers (torch.Tensor): Defined depth layers used for
segmenting the depth map.
        use_nonlinear (bool): Flag indicating whether to apply non-
linear processing.
        is_blender (bool): Flag indicating if depth files are in
Blender's EXR format.
        scale_factor (float): Scale factor used for normalizing the
depth values.

    """
```

```
    # Define paths for the depth and image folders within the root
directory.
    depth_folder = os.path.join(root, "depth")
    image_folder = os.path.join(root, "rgb")
    os.environ["OPENCV_IO_ENABLE_OPENEXR"] = "1"

    # Create the output folder where processed images will be saved.
    output_folder = os.path.join(root, "Codedphasecam-27Linear-New")
    os.makedirs(output_folder, exist_ok=True)  # Ensure the folder
exists, create it if it doesn't.

    # Get the list of files in the image folder.
    files = os.listdir(image_folder)

    # Variable to keep track of the maximum depth value found across
all depth maps.
    max_depth_value = 0

    # Padding size is half the width/height of the PSFs (assuming
square filters).
    padding = psfs.shape[2] // 2

    # Loop through each file in the image folder, displaying progress
using tqdm.
    for idx, file in tqdm.tqdm(enumerate(files), total=len(files),
desc=root):

        # Construct the full paths for the image and corresponding
depth files.
        image_file = os.path.join(image_folder, file)
        depth_file = os.path.join(depth_folder, file).replace(".png",
".exr") if is_blender else os.path.join(depth_folder, file)

        # Read the image file (in BGR format since OpenCV loads in BGR
by default).
        image_bgr = cv2.imread(image_file)

        # Convert the image from BGR to RGB format (as needed for
processing).
        image = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)

        # Read the depth file, handling both EXR (Blender) and other
formats.
        raw_depth = cv2.imread(depth_file, cv2.IMREAD_ANYCOLOR |
cv2.IMREAD_ANYDEPTH)

        # Check if the depth file is missing or unreadable, skip
processing for that file if so.
        if raw_depth is None:
            print(f"{file} is missing a depth file")
```

```python
            continue

        # If using Blender's EXR format, extract the depth from the first channel.
        # Otherwise, normalize the depth using the provided scale factor.
        metric_depth = raw_depth[:, :, 0] if is_blender else raw_depth / scale_factor

        # Apply depth-dependent convolution to the image, either linear or non-linear depending on the flag.
        coded_image_rgb = capture_image(image.astype(np.float32), metric_depth, psfs, depth_layers, padding)

        # Convert the processed image back to BGR format for saving (as OpenCV saves in BGR by default).
        coded_image_bgr = cv2.cvtColor(coded_image_rgb, cv2.COLOR_RGB2BGR)

        # Save the processed image to the output folder.
        cv2.imwrite(os.path.join(output_folder, file), coded_image_bgr)

        # Update the maximum depth value encountered in this folder.
        max_depth_value = max(max_depth_value, np.max(metric_depth))

    # Print the maximum depth value found in the folder for reference.
    print(f"Max Depth Value in the folder: {max_depth_value}")
```

Main Function:

Put everything together! Modify the Image and PSF file path according to your needs.

Note that scale_factor relies on the camera hardware and is different for different datasets.

Note: Our Point Spread Functions (PSFs) correspond to discretized depth layers using a 23×23 Zernike parameterized phase mask,with the depth range discretized into 27 bins within the interval of [0.5, 6] meters, with a focal distance of 85 cm.

```python
def main():
    # Define and configure an argument parser (currently commented out).
    # The argument parser allows users to specify the root directory, whether the depth images are in Blender's EXR format,
    # and the scale factor for depth normalization through command-line arguments.
    parser = argparse.ArgumentParser(description="Process images with depth-dependent processing.")
```

```
    parser.add_argument("--root", type=str, required=True, help="Root
directory containing the datasets.")
    parser.add_argument("--is_blender", action="store_true", help="Use
Blender's EXR depth format.")
    parser.add_argument("--scale_factor", type=float, default=5000,
help="Scale factor for depth normalization.")
    args = parser.parse_args(["--root", "C:/Adv Computer
Vision/P2/UMD-CodedVO-dataset/DiningRoom/","--is_blender","--
scale_factor", "1"])
    #Following arguments are for running Livingroom and NYU Data
    # args = parser.parse_args(["--root", "C:/Adv Computer
Vision/P2/train/LivingRoom1/","--is_blender","--scale_factor", "1"])
    # args = parser.parse_args(["--root", "C:/Adv Computer
Vision/P2/train/nyu_data/nyu_data/", "--scale_factor", "1000"])


    # Print the device being used (e.g., GPU or CPU), which was set
earlier in the code using `torch.device()`.
    print(device)
    # Define the root directory where the image and depth datasets are
located.
    # This is hardcoded in the script but could be passed as an
argument using the argument parser (currently commented).
    # root = '/content/drive/MyDrive/train/LivingRoom1'

    root = args.root  # Root directory from arguments
    is_blender = args.is_blender  # Whether to use Blender's EXR depth
format
    scale_factor = args.scale_factor
    # root = 'C:/Adv Computer
Vision/P2/UMD-CodedVO-dataset/DiningRoom/'

    # Flag indicating whether the depth maps are in Blender's EXR
format.
    # is_blender = True

    # Define the scale factor for depth normalization (used when the
depth maps are not in EXR format).
    # scale_factor = 5000 # This scale is different for different
datasets
    # It is 1000 for NYUv2 dataset
    # It is 5000 for ICL-NUIM dataset
    # It is 1 for all other dataset provided (UMD-CodedVO/DiningRoom
etc. datasets)
    # Depth sensors and datasets typically store depth as integer or
floating-point
    # values. However, these values are often not in direct metric
units (like meters)
    # because it would require storing large floating-point values,
    # which increases storage size and precision issues.
```

```python
    # Load the Point Spread Functions (PSFs) from a `.npy` file.
    # psf_path = '/content/drive/MyDrive/phasecam-psfs-27.npy'
    psf_path = './phasecam-psfs-27.npy'

    # Define the depth layers, which divide the depth map into segments based on depth values.
    # `torch.linspace` creates a tensor with 27 evenly spaced depth layers between 0.5 and 6 units of depth.

########################################################################
    # TODO: depth_layers = ...

########################################################################
    depth_layers = torch.linspace(0.5, 6.0, 27).to(device)
    # Load the PSFs from the `.npy` file and adjust their axis order using `np.moveaxis`.
    # This is done to match the format expected by the convolution function.
    psfs = torch.from_numpy(np.moveaxis(np.load(psf_path), -1, 1))

    # Call the `process_folder` function to process all images in the folder.
    # The parameters passed include the root directory, loaded PSFs, defined depth layers,
    # a flag for whether to use non-linear processing, the Blender depth format flag, and the scale factor.
    process_folder(
        root=root,
        psfs=psfs,
        depth_layers=depth_layers,
        is_blender=is_blender,
        use_nonlinear = False,# Use Blender's EXR depth format if set to True.
        scale_factor=scale_factor,     # The scale factor for depth normalization.
    )

# Run the main function when the script is executed.
main()
```

```
cuda

C:/Adv Computer Vision/P2/UMD-CodedVO-dataset/DiningRoom/: 100%|██████████| 1000/1000 [04:31<00:00,  3.69it/s]

Max Depth Value in the folder: 65504.0
```

Submission Guidelines:

You are required to submit:

1. The compiled version of this colab file with results
2. PDF Report (Combined with Part 2)
3. Final Coded Image Output using Linear Convolution on the following image set (Images: '1.png', '10.png', '20.png', '30.png', '40.png')

   To confirm your results are correct, compare the results of your linear model with the following images: https://drive.google.com/drive/folders/1m0ihEGnDnOtedTpI8VRJ8miD25Bxqxa-?usp=sharing
   (Note that: Do not compare with Non-Linear nyu_data output given here: https://drive.google.com/drive/folders/12GrDxTBMaSlGeMRWycxmCQl01BHnC5-O)

Note: Except the pixels in the final coded image that are far away to be black something like this: https://drive.google.com/file/d/14Kg2BCs-Mie5Y-udDanNJPAz0PGae_nE/view?usp=sharing