ECE7106 Robot Vision – Term Project

22241324
윤주현

- Youtube : https://youtu.be/x-Ns2UA3Ozg

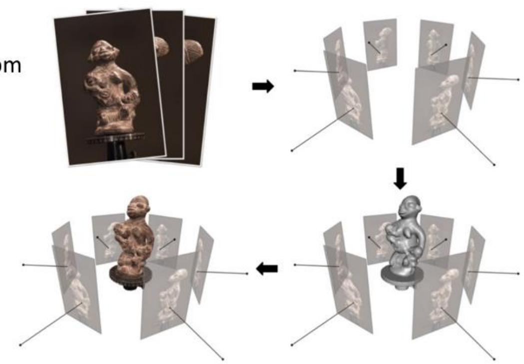
Steps

- Approach
- Single Camera Setup
- Capture Multi-View Images
- Prepare the Data
- Landmark detection
- Blending texture
- Skin estimation
- Reconstruction
- Rendering
- Libraries, computer specifications

❖ **Multi-View Stereo**

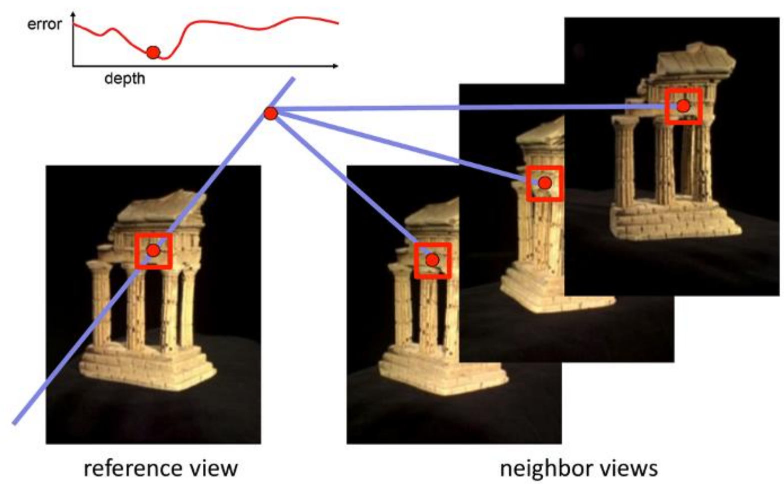
Multi-view Stereo

- **Input**
  - Calibrated N images from several viewpoints
  - Known intrinsics, extrinsics, projection matrices
- **Output**
  - 3D object model



Clockwise: input imagery, posed imagery, reconstructed 3D geometry, textured 3D geometry

Multi-View Stereo: Basic Idea



Capture Multi-View Images

Single Camera Setup

- just one camera: iphone 12
- Subject Positioning: Placed my face in a static pose, minimal movement during capture.
- Lighting: Used diffuse lighting to avoid harsh shadows, maintained consistent lighting across all image s.
- Background: Used a plain, non-reflective background to simplify segmentation in post-processing.

AUTONOMOUS NAVIGATION LAB

❖ Capture Multi-View Images

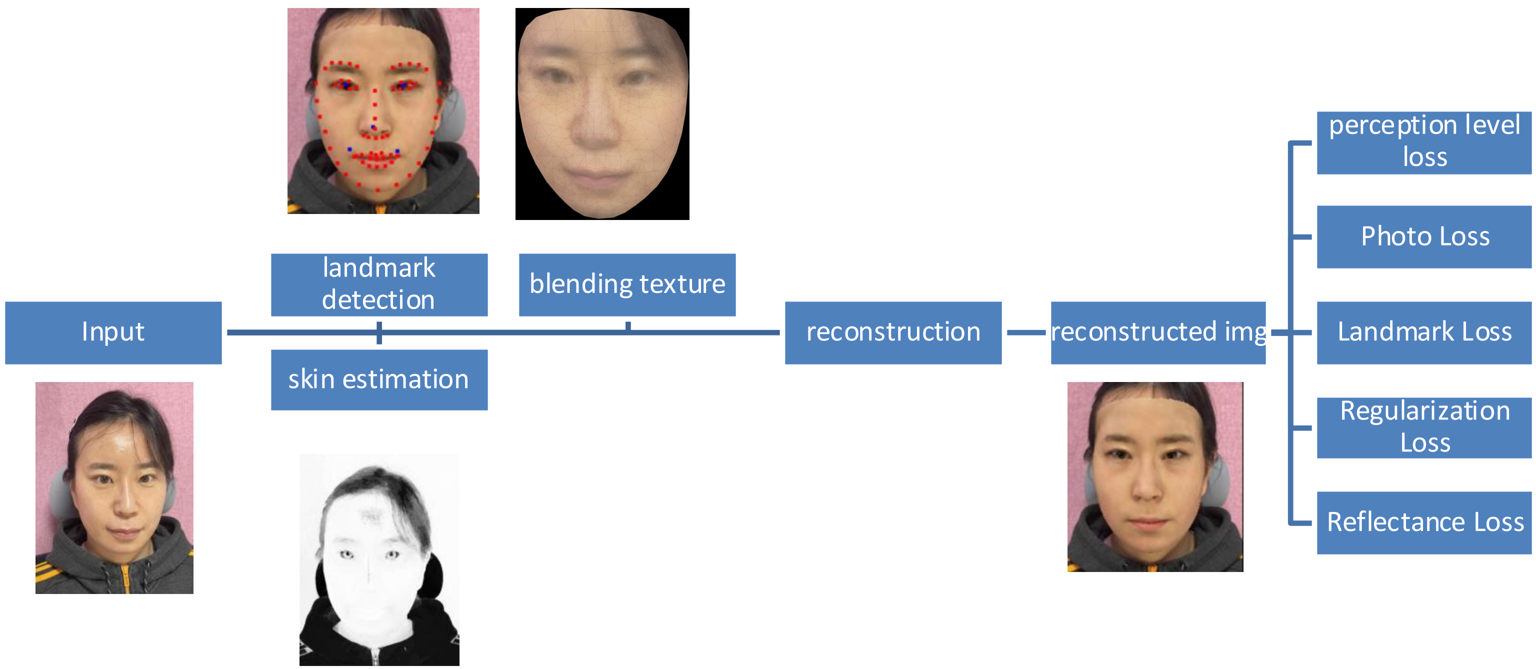
- ➢ Reference View: Started with a frontal image of my face as the reference view. This view is critical as i t acts as the baseline for matching features in other views.
- ➢ Neighbor Views: Captured 52 images of my face from different angles around it.
  - Spacing: Covered 180° around my face in increments of 15-20°.
  - Angles: Included slight variations in vertical angles (tilt the camera slightly up and down) to capture 3D depth effectively.
- ➢ Overlap: overlapped 60-80% between consecutive images.
- ➢ Capture Consistency: Maintained consistent camera focus, exposure, and distance across all views.

Prepare the Data

- Organize Images: Renamed and numbered images sequentially from HEIC to jpg.

- Clean the Data: Removed blurry or inconsistent images.
- Camera Calibration: computed intrinsic parameters (focal length, distortion) using with chessboard ca libration images by OpenCV

workflow


Input
landmark detection
skin estimation
blending texture
reconstruction
reconstructed img
perception level loss
Photo Loss
Landmark Loss
Regularization Loss
Reflectance Loss

❖ landmark detection workflow in Code

input

- The image is loaded using OpenCV

detection

- it processes the image and identifies faces
- keypoints corresponding to the five landmarks for each face

landmark extraction

- The keypoints are extracted from the first detected face(Reference view)
- stored in a structured format, such as a 5x2 numpy array, where each row represents a landmark's (x, y) coordinates.

output

```python
import cv2
import numpy as np


# Paths
dataset_path = './datasets/training_images'
detections_path = os.path.join(dataset_path, 'detections')
os.makedirs(detections_path, exist_ok=True)

def generate_landmarks(image_path, output_path):
    # Load the image
    img = cv2.imread(image_path)

    results = detector.detect_faces(img)

    if not results:
        print(f"No faces detected in {image_path}")
        return

    # Get the first face's keypoints (assuming single face per image)
    keypoints = results[0]['keypoints']
    landmarks = np.array([
        keypoints['left_eye'],
        keypoints['right_eye'],
        keypoints['nose'],
        keypoints['mouth_left'],
        keypoints['mouth_right']
    ])

    # Save the landmarks to a text file
    np.savetxt(output_path, landmarks, fmt="%.2f")
    print(f"Landmarks saved for {image_path} -> {output_path}")

# Process all images
for img_file in os.listdir(dataset_path):
    if img_file.lower().endswith(('.png', '.jpg', '.jpeg')):
        img_path = os.path.join(dataset_path, img_file)
        txt_path = os.path.join(detections_path, os.path.splitext(img_file)[0] + '.txt')
        generate_landmarks(img_path, txt_path)

print("Dataset preparation completed.")
```

❖ landmark detection workflow in Code

- datasets
  - training_images
    - detections
      - IMG_8329.txt U
      - IMG_8330.txt U
      - IMG_8331.txt U
      - IMG_8332.txt U
      - IMG_8340.txt U
      - IMG_8341.txt U
      - IMG_8342.txt U
      - IMG_8343.txt U
      - IMG_8344.txt U
      - IMG_8345.txt U
      - IMG_8346.txt U
      - IMG_8354.txt U
      - IMG_8355.txt U
      - IMG_8356.txt U
      - IMG_8365.txt U
      - IMG_8366.txt U
      - IMG_8367.txt U
      - IMG_8368.txt U
      - IMG_8373.txt U
      - IMG_8374.txt U
      - IMG_8375.txt U
      - IMG_8376.txt U
      - IMG_8377.txt U

```
896.00 1921.00
1752.00 1940.00
1293.00 2557.00
953.00 2893.00
1659.00 2924.00
```

Blending texture

❖ Blending texture workflow in Code

input
- Images and their landmarks.

Triangulation
- Generate triangles based on the base image's landmarks.

Warping
- Align textures from other images using affine transformation.

Blending
- Accumulate and average warped textures to produce the final blended texture.

```python
def warp_image(base_image, target_image, base_landmarks, target_landmarks):
    base_landmarks_2d = base_landmarks[:, :2]
    target_landmarks_2d = target_landmarks[:, :2]

    rect = (0, 0, base_image.shape[1], base_image.shape[0])
    subdiv = cv2.Subdiv2D(rect)
    for x, y in base_landmarks_2d:
        subdiv.insert((x, y))
    triangles = subdiv.getTriangleList()

    warped_image = np.zeros_like(base_image)
    for t in triangles:
        base_pts = np.array([[t[0], t[1]], [t[2], t[3]], [t[4], t[5]]], np.float32)
        target_pts = [target_landmarks_2d[np.argmin(np.linalg.norm(base_landmarks_2d - bp, axis=1))] for bp in base_pts]
        affine_matrix = cv2.getAffineTransform(np.array(target_pts, np.float32), base_pts)
        triangle_mask = np.zeros_like(base_image)
        cv2.fillConvexPoly(triangle_mask, np.int32(base_pts), (1, 1, 1))
        warped_triangle = cv2.warpAffine(target_image, affine_matrix, (base_image.shape[1], base_image.shape[0]))
        warped_image += warped_triangle * triangle_mask

    return warped_image
```

```python
base_image_path = os.pat (variable) base_image_path: str ")
base_image = cv2.imread(base_image_path)
final_texture = np.zeros_like(base_image, dtype=np.float32)
```

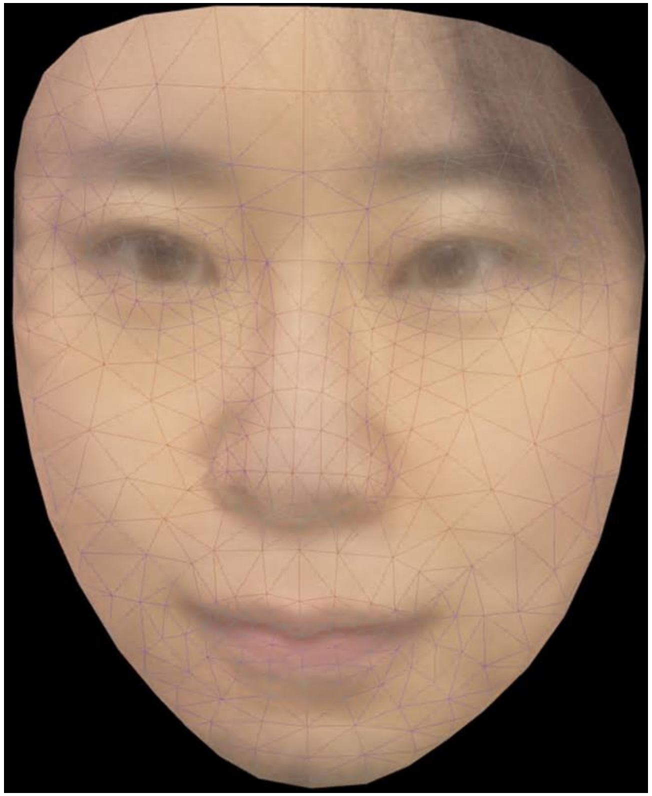
```python
image_files = sorted([f for f in os.listdir(input_folder) if f.endswith(".jpg")])
landmarks_files = sorted([f for f in os.listdir(output_folder) if f.endswith(".txt")])
valid_pairs = [(img_file, f"{img_file.split('.')[0]}_landmarks.txt") for img_file in image_files if f"{img_file.split('.')[0]}_landmarks.txt" in landmarks_files]

print("Blending textures...")
for img_file, lmk_file in tqdm(valid_pairs, desc="Warping images"):
    target_image = cv2.imread(os.path.join(input_folder, img_file))
    base_landmarks = np.loadtxt(os.path.join(output_folder, valid_pairs[0][1]))
    target_landmarks = np.loadtxt(os.path.join(output_folder, lmk_file))
    warped_image = warp_image(base_image, target_image, base_landmarks, target_landmarks)
    final_texture += warped_image.astype(np.float32)
```

```
Blending textures...
Warping images: 100%|██████████| 33/33 [20:19<00:00, 36.95s/it]
```

```python
# Load landmarks and texture
vertices = np.loadtxt("./landmarks.txt")
texture_image = cv2.imread("C:/Users/User/Workspace/Robot_vision/blended_texture.jpg")
texture_image = cv2.cvtColor(texture_image, cv2.COLOR_BGR2RGB)
# texture_image = cv2.resize(texture_image, (512, 512))  # Optimize texture size
texture_image = cv2.resize(texture_image, (256, 256))  # Resize to 256x256


# Normalize Z-values to prevent spikes
# vertices[:, 2] = (vertices[:, 2] - np.min(vertices[:, 2])) / (np.max(vertices[:, 2]) - np.min(vertices[:, 2])) * 10
vertices = vertices[:5000]  # Limit to 10,000 vertices for testing


# Start profiling
start_time = time.time()

# Generate Delaunay triangulation on XY coordinates
tri = Delaunay(vertices[:, :2])
print("Delaunay triangulation time:", time.time() - start_time)

# Normalize UV coordinates for texture mapping
uv_coords = vertices[:, :2].copy()
uv_coords[:, 0] /= np.max(vertices[:, 0])  # Normalize X
uv_coords[:, 1] /= np.max(vertices[:, 1])  # Normalize Y
uv_coords[:, 1] = 1 - uv_coords[:, 1]      # Flip Y-axis

# Create TriangleMesh
mesh = o3d.geometry.TriangleMesh()


tart_time = time.time()
mesh.vertices = o3d.utility.Vector3dVector(vertices)
print("Vertex assignment time:", time.time() - start_time)
```

Blending texture

Aspect	Explanation
Purpose	To blend textures from multiple facial images into a single smooth texture for better appearance and consistency in 3D f ace reconstruction.
Landmarks	Used to align corresponding features (eyes, nose, mouth, etc.) between images for accurate blending.
Delaunay Triangulation	Divides the face into triangles based on landmarks, ensuring smooth warping of textures without distortions.
Affine Transformation	Warps individual triangles from the target_image to align with the corresponding triangles in the base_image.
Blending Strategy	Accumulates the warped textures from all images and averages them to reduce noise.
Final Texture	Represents the average texture of the face across all input images.



❖ skin estimation workflow in Code

- Gaussian Mixture Model (GMM)
  - It differentiates between skin and non-ski n regions in the image.
  - It is a probabilistic model that represents t he distribution of data as a mixture of mul tiple Gaussian components.

```python
import math
import numpy as np
import os
import cv2

class GMM:
    def __init__(self, dim, num, w, mu, cov, cov_det, cov_inv):
        self.dim = dim # feature dimension
        self.num = num # number of Gaussian components
        self.w = w # weights of Gaussian components (a list of scalars)
        self.mu= mu # mean of Gaussian components (a list of 1xdim vectors)
        self.cov = cov # covariance matrix of Gaussian components (a list of dimxdim matrices)
        self.cov_det = cov_det # pre-computed determinet of covariance matrices (a list of scalars)
        self.cov_inv = cov_inv # pre-computed inverse covariance matrices (a list of dimxdim matrices)

        self.factor = [0]*num
        for i in range(self.num):
            self.factor[i] = (2*math.pi)**(self.dim/2) * self.cov_det[i]**0.5

    def likelihood(self, data):
        assert(data.shape[1] == self.dim)
        N = data.shape[0]
        lh = np.zeros(N)

        for i in range(self.num):
            data_ = data - self.mu[i]

            tmp = np.matmul(data_,self.cov_inv[i]) * data_
            tmp = np.sum(tmp,axis=1)
            power = -0.5 * tmp

            p = np.array([math.exp(power[j]) for j in range(N)])
            p = p/self.factor[i]
            lh += p*self.w[i]

        return lh
```

AUTONOMOUS NAVIGATION LAB

skin estimation workflow in Code

- Color Space Conversion
  - The input image is converted from BGR to YCbCr color space.
    - YCbCr is a perceptual color space
    - Y: Luminance (brightness).
    - Cb: Chrominance-blue (color difference).
    - Cr: Chrominance-red.
  - The Cb and Cr channels are particularly effective for distinguishing skin tones.

```python
def _rgb2ycbcr(rgb):
    m = np.array([[65.481, 128.553, 24.966],
                  [-37.797, -74.203, 112],
                  [112, -93.786, -18.214]])
    shape = rgb.shape
    rgb = rgb.reshape((shape[0] * shape[1], 3))
    ycbcr = np.dot(rgb, m.transpose() / 255.)
    ycbcr[:, 0] += 16.
    ycbcr[:, 1:] += 128.
    return ycbcr.reshape(shape)


def _bgr2ycbcr(bgr):
    rgb = bgr[..., ::-1]
    return _rgb2ycbcr(rgb)
```

skin estimation

❖ skin estimation workflow in Code

➢ Parameters for Skin and Non-Skin

- gmm_skin: skin regions.
- gmm_nonskin: non-skin regions.
- Weights (w): Importance of each Gaussian component.
- Means (mu): The average feature values (e.g., colors) for each Gaussian component.
- Covariances (cov): The variance of feature values for each component, defining its spread.

```
gmm_skin_w = [0.24063933, 0.16365987, 0.26034665, 0.33535415]
gmm_skin_mu = [np.array([113.71862, 103.39613, 164.08226]),
                np.array([150.19858, 105.18467, 155.51428]),
                np.array([183.92976, 107.62468, 152.71820]),
                np.array([114.90524, 113.59782, 151.38217])]
gmm_skin_cov_det = [5692842.5, 5851930.5, 2329131., 1585971.]
gmm_skin_cov_inv = [np.array([[0.0019472069, 0.0020450759, -0.00060243998],[0.0020450759, 0.017700525, 0.0051420014],[-0.00060243998, 0.0051420014, 0.0081308950]]),
                    np.array([[0.0027110141, 0.0011036990, 0.0023122299],[0.0011036990, 0.010707724, 0.010742856],[0.0023122299, 0.010742856, 0.017481629]]),
                    np.array([[0.0048026871, 0.00022935172, 0.0077668377],[0.00022935172, 0.011729696, 0.0081661865],[0.0077668377, 0.0081661865, 0.025374353]]),
                    np.array([[0.0011989699, 0.0022453172, -0.0010748957],[0.0022453172, 0.047758564, 0.020332102],[-0.0010748957, 0.020332102, 0.024502251]])]

gmm_skin = GMM(3, 4, gmm_skin_w, gmm_skin_mu, [], gmm_skin_cov_det, gmm_skin_cov_inv)

gmm_nonskin_w = [0.12791070, 0.31130761, 0.34245777, 0.21832393]
gmm_nonskin_mu = [np.array([99.200851, 112.07533, 140.20602]),
                  np.array([110.91392, 125.52969, 130.19237]),
                  np.array([129.75864, 129.96107, 126.96808]),
                  np.array([112.29587, 128.85121, 129.05431])]
gmm_nonskin_cov_det = [458703648., 6466488., 90611376., 133097.63]
gmm_nonskin_cov_inv = [np.array([[0.00085371657, 0.00071197288, 0.00023958916],[0.00071197288, 0.0025935620, 0.00076557708],[0.00023958916, 0.00076557708, 0.0015042332]]),
                       np.array([[0.00024650150, 0.00045542428, 0.00015019422],[0.00045542428, 0.026412144, 0.018419769],[0.00015019422, 0.018419769, 0.037497383]]),
                       np.array([[0.00037054974, 0.00038146760, 0.00040408765],[0.00038146760, 0.0085505722, 0.0079136286],[0.00040408765, 0.0079136286, 0.010982352]]),
                       np.array([[0.00013709733, 0.00051228428, 0.00012777430],[0.00051228428, 0.28237113, 0.10528370],[0.00012777430, 0.10528370, 0.23468947]])]

gmm_nonskin = GMM(3, 4, gmm_nonskin_w, gmm_nonskin_mu, [], gmm_nonskin_cov_det, gmm_nonskin_cov_inv)

prior_skin = 0.8
prior_nonskin = 1 - prior_skin
```

AUTONOMOUS NAVIGATION LAB

❖ skin estimation workflow in Code

Skin and Non-Skin Likelihoods

- The likelihood of it belonging to skin is calculated using
- The likelihood of it belonging to non-skin is calculated using

Posterior Probability Calculation

- $P(\text{skin}|\text{pixel}) = \frac{P(\text{pixel}|\text{skin})\cdot P(\text{skin})}{P(\text{pixel}|\text{skin})\cdot P(\text{skin})+P(\text{pixel}|\text{non-skin})\cdot P(\text{non-skin})}$

Generating the Skin Mask

- The posterior probability values are scaled to 0–255 and reshaped to match the image dimensions.
- The result is a grayscale mask, where higher intensity values represent higher probabilities of being skin.

```python
# calculate skin attention mask
def skinmask(imbgr):
    im = _bgr2ycbcr(imbgr)

    data = im.reshape((-1,3))

    lh_skin = gmm_skin.likelihood(data)
    lh_nonskin = gmm_nonskin.likelihood(data)

    tmp1 = prior_skin * lh_skin
    tmp2 = prior_nonskin * lh_nonskin
    post_skin = tmp1 / (tmp1+tmp2) # posterior probability

    post_skin = post_skin.reshape((im.shape[0],im.shape[1]))

    post_skin = np.round(post_skin*255)
    post_skin = post_skin.astype(np.uint8)
    post_skin = np.tile(np.expand_dims(post_skin,2),[1,1,3]) # reshape to H*W*3

    return post_skin


def get_skin_mask(img_path):
    print('generating skin masks......')
    names = [i for i in sorted(os.listdir(
        img_path)) if 'jpg' in i or 'png' in i or 'jpeg' in i or 'PNG' in i]
    save_path = os.path.join(img_path, 'mask')
    if not os.path.isdir(save_path):
        os.makedirs(save_path)

    for i in range(0, len(names)):
        name = names[i]
        print('%05d' % (i), ' ', name)
        full_image_name = os.path.join(img_path, name)
        img = cv2.imread(full_image_name).astype(np.float32)
        skin_img = skinmask(img)
        cv2.imwrite(os.path.join(save_path, name), skin_img.astype(np.uint8))
```

skin estimation

❖ skin estimation

reconstruction

Loss Function	Related Coefficients	Purpose
Perceptual Loss	α	Matches the identity feature (cosine similarity).
Photo Loss	δ,γ	Matches texture and lighting to the target image.
Landmark Loss	p,β	Aligns landmarks by adjusting pose and expression.
Regularization Loss	α,β,δ,γ	Regularizes plausible identity, expression, texture, and lighting.
Reflectance Loss	δ	Enforces uniform albedo for consistent texture across the face surface.

```python
import numpy as np
import torch
import torch.nn as nn
from kornia.geometry import warp_affine
import torch.nn.functional as F

def resize_n_crop(image, M, dsize=112):
    # image: (b, c, h, w)
    # M   :  (b, 2, 3)
    return warp_affine(image, M, dsize=(dsize, dsize))

### perceptual level loss
class PerceptualLoss(nn.Module):
    def __init__(self, recog_net, input_size=112):
        super(PerceptualLoss, self).__init__()
        self.recog_net = recog_net
        self.preprocess = lambda x: 2 * x - 1
        self.input_size=input_size
    def forward(imageA, imageB, M):
        """
        1 - cosine distance
        Parameters:
            imageA       --torch.tensor (B, 3, H, W), range (0, 1) , RGB order
            imageB       --same as imageA
        """

        imageA = self.preprocess(resize_n_crop(imageA, M, self.input_size))
        imageB = self.preprocess(resize_n_crop(imageB, M, self.input_size))

        # freeze bn
        self.recog_net.eval()

        id_featureA = F.normalize(self.recog_net(imageA), dim=-1, p=2)
        id_featureB = F.normalize(self.recog_net(imageB), dim=-1, p=2)
        cosine_d = torch.sum(id_featureA * id_featureB, dim=-1)
        # assert torch.sum((cosine_d > 1).float()) == 0
        return torch.sum(1 - cosine_d) / cosine_d.shape[0]

def perceptual_loss(id_featureA, id_featureB):
    cosine_d = torch.sum(id_featureA * id_featureB, dim=-1)
        # assert torch.sum((cosine_d > 1).float()) == 0
    return torch.sum(1 - cosine_d) / cosine_d.shape[0]
```

```python
### image level loss
def photo_loss(imageA, imageB, mask, eps=1e-6):
    """
    l2 norm (with sqrt, to ensure backward stabililty, use eps, otherwise Nan may occur)
    Parameters:
        imageA       --torch.tensor (B, 3, H, W), range (0, 1), RGB order
        imageB       --same as imageA
    """
    loss = torch.sqrt(eps + torch.sum((imageA - imageB) ** 2, dim=1, keepdims=True)) * mask
    loss = torch.sum(loss) / torch.max(torch.sum(mask), torch.tensor(1.0).to(mask.device))
    return loss

def landmark_loss(predict_lm, gt_lm, weight=None):
    """
    weighted mse loss
    Parameters:
        predict_lm    --torch.tensor (B, 68, 2)
        gt_lm         --torch.tensor (B, 68, 2)
        weight        --numpy.array (1, 68)
    """
    if not weight:
        weight = np.ones([68])
        weight[28:31] = 20
        weight[-8:] = 20
        weight = np.expand_dims(weight, 0)
        weight = torch.tensor(weight).to(predict_lm.device)
    loss = torch.sum((predict_lm - gt_lm)**2, dim=-1) * weight
    loss = torch.sum(loss) / (predict_lm.shape[0] * predict_lm.shape[1])
    return loss
```

```python
### regulization
def reg_loss(coeffs_dict, opt=None):

    # coefficient regularization to ensure plausible 3d faces
    if opt:
        w_id, w_exp, w_tex = opt.w_id, opt.w_exp, opt.w_tex
    else:
        w_id, w_exp, w_tex = 1, 1, 1, 1
    creg_loss = w_id * torch.sum(coeffs_dict['id'] ** 2) +  \
           w_exp * torch.sum(coeffs_dict['exp'] ** 2) + \
           w_tex * torch.sum(coeffs_dict['tex'] ** 2)
    creg_loss = creg_loss / coeffs_dict['id'].shape[0]

    # gamma regularization to ensure a nearly-monochromatic light
    gamma = coeffs_dict['gamma'].reshape([-1, 3, 9])
    gamma_mean = torch.mean(gamma, dim=1, keepdims=True)
    gamma_loss = torch.mean((gamma - gamma_mean) ** 2)

    return creg_loss, gamma_loss

def reflectance_loss(texture, mask):
    """
    minimize texture variance (mse), albedo regularization to ensure an uniform skin albedo
    Parameters:
        texture       --torch.tensor, (B, N, 3)
        mask          --torch.tensor, (N), 1 or 0

    """
    mask = mask.reshape([1, mask.shape[0], 1])
    texture_mean = torch.sum(mask * texture, dim=1, keepdims=True) / torch.sum(mask)
    loss = torch.sum(((texture - texture_mean) * mask)**2) / (texture.shape[0] * torch.sum(mask))
    return loss
```

reconstruction

Component	Functionality	Equation	Purpose
3D Shape (S)	Computes the 3D shape	$S = \bar{S} + B_{\mathrm{id}} \cdot \alpha + B_{\mathrm{exp}} \cdot \beta$	Generates the 3D geometry of the face, incl uding individual identity and expressions.
Texture (T)	Computes the vertex text ure using the texture coef ficients (δ)	$T = \bar{T} + B_{\mathrm{tex}} \cdot \delta$	Generates the texture or color information for each vertex of the 3D face geometry.

```python
def compute_shape(self, id_coeff, exp_coeff):
    """
    Return:
        face_shape       -- torch.tensor, size (B, N, 3)

    Parameters:
        id_coeff         -- torch.tensor, size (B, 80), identity coeffs
        exp_coeff        -- torch.tensor, size (B, 64), expression coeffs
    """
    batch_size = id_coeff.shape[0]
    id_part = torch.einsum('ij,aj->ai', self.id_base, id_coeff)
    exp_part = torch.einsum('ij,aj->ai', self.exp_base, exp_coeff)
    face_shape = id_part + exp_part + self.mean_shape.reshape([1, -1])
    return face_shape.reshape([batch_size, -1, 3])


def compute_texture(self, tex_coeff, normalize=True):
    """
    Return:
        face_texture     -- torch.tensor, size (B, N, 3), in RGB order, range (0, 1.)

    Parameters:
        tex_coeff        -- torch.tensor, size (B, 80)
    """
    batch_size = tex_coeff.shape[0]
    face_texture = torch.einsum('ij,aj->ai', self.tex_base, tex_coeff) + self.mean_tex
    if normalize:
        face_texture = face_texture / 255.
    return face_texture.reshape([batch_size, -1, 3])
```

reconstruction

rendering

Component	Functionality	Equation
Rendering	Combines 3D shape (S) and texture (T)	Rendered Image= Renderer(S,T,Camera Parameters)

```python
def ndc_projection(x=0.1, n=1.0, f=50.0):
    return np.array([[n/x,    0,            0,              0],
                     [  0, n/-x,            0,              0],
                     [  0,    0, -(f+n)/(f-n), -(2*f*n)/(f-n)],
                     [  0,    0,           -1,              0]]).astype(np.float32)

class MeshRenderer(nn.Module):
    def __init__(self,
                rasterize_fov,
                znear=0.1,
                zfar=10,
                rasterize_size=224,
                use_opengl=True):
        super(MeshRenderer, self).__init__()

        x = np.tan(np.deg2rad(rasterize_fov * 0.5)) * znear
        self.ndc_proj = torch.tensor(ndc_projection(x=x, n=znear, f=zfar)).matmul(
                torch.diag(torch.tensor([1., -1, -1, 1])))
        self.rasterize_size = rasterize_size
        self.use_opengl = use_opengl
        self.ctx = None

    def forward(self, vertex, tri, feat=None):
        """
        Return:
            mask               -- torch.tensor, size (B, 1, H, W)
            depth              -- torch.tensor, size (B, 1, H, W)
            features(optional) -- torch.tensor, size (B, C, H, W) if feat is not None

        Parameters:
            vertex          -- torch.tensor, size (B, N, 3)
            tri             -- torch.tensor, size (B, M, 3) or (M, 3), triangles
            feat(optional)  -- torch.tensor, size (B, C), features
        """
        device = vertex.device
        rsize = int(self.rasterize_size)
        ndc_proj = self.ndc_proj.to(device)
        # trans to homogeneous coordinates of 3d vertices, the direction of y is the same as v
        if vertex.shape[-1] == 3:
            vertex = torch.cat([vertex, torch.ones([*vertex.shape[:2], 1]).to(device)], dim=-1)
            vertex[..., 1] = -vertex[..., 1]
```

rendering

AUTONOMOUS NAVIGATION LAB

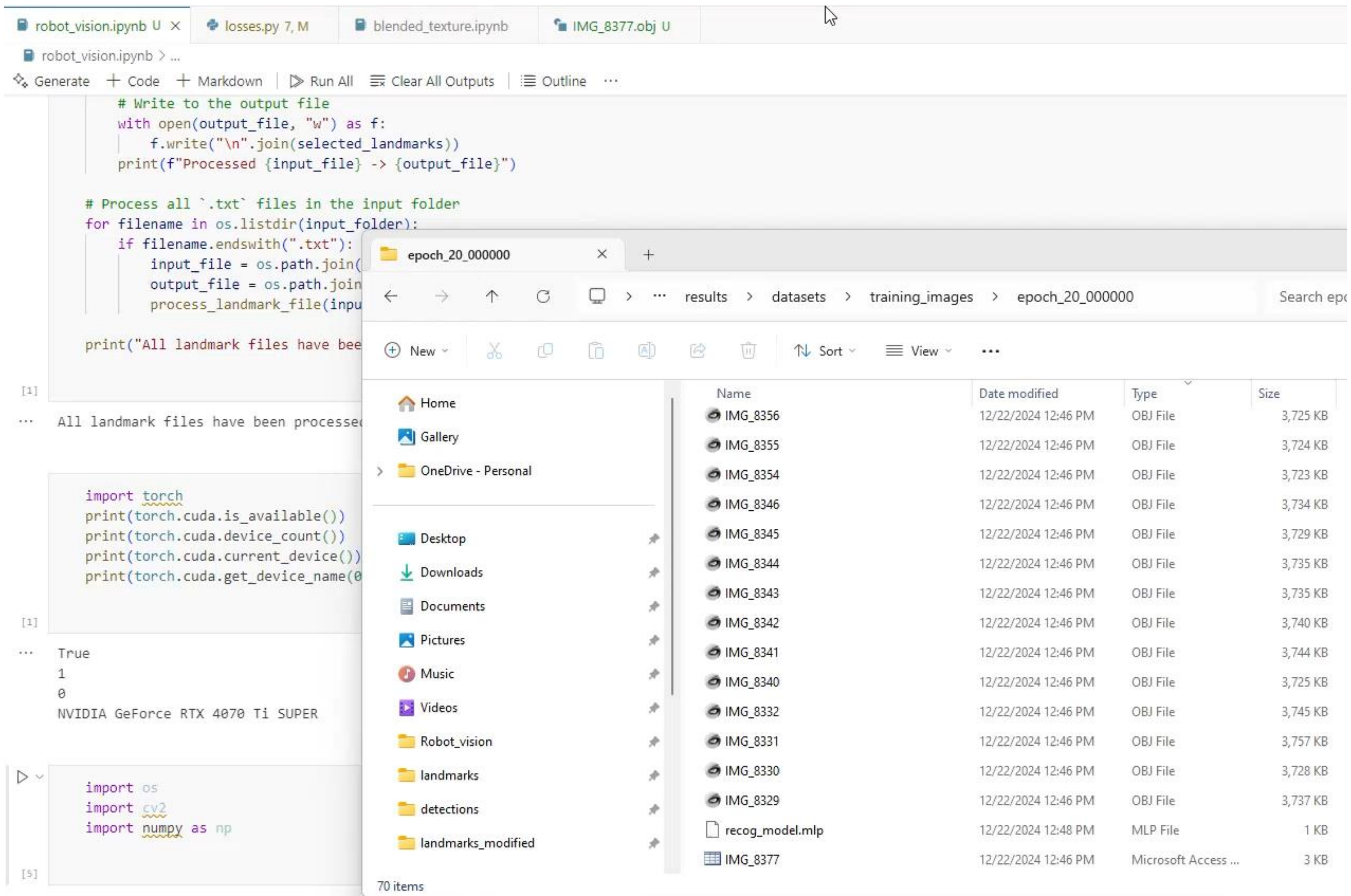
➢ Generate OBJ for rendering

```
vertex_ndc = vertex @ ndc_proj.t()
if self.ctx is None:
    if self.use_opengl:
        self.ctx = dr.RasterizeGLContext(device=device)
        ctx_str = "opengl"
    else:
        self.ctx = dr.RasterizeCudaContext(device=device)
        ctx_str = "cuda"
    print("create %s ctx on device cuda:%d"%(ctx_str, device.index))

ranges = None
if isinstance(tri, List) or len(tri.shape) == 3:
    vum = vertex_ndc.shape[1]
    fnum = torch.tensor([f.shape[0] for f in tri]).unsqueeze(1).to(device)
    fstartidx = torch.cumsum(fnum, dim=0) - fnum
    ranges = torch.cat([fstartidx, fnum], axis=1).type(torch.int32).cpu()
    for i in range(tri.shape[0]):
        tri[i] = tri[i] + i*vum
    vertex_ndc = torch.cat(vertex_ndc, dim=0)
    tri = torch.cat(tri, dim=0)

# for range_mode vetex: [B*N, 4], tri: [B*M, 3], for instance_mode vetex: [B, N, 4], tri: [M, 3]
tri = tri.type(torch.int32).contiguous()
rast_out, _ = dr.rasterize(self.ctx, vertex_ndc.contiguous(), tri, resolution=[rsize, rsize], ranges=ranges)

depth, _ = dr.interpolate(vertex.reshape([-1,4])[...,2].unsqueeze(1).contiguous(), rast_out, tri)
depth = depth.permute(0, 3, 1, 2)
mask =  (rast_out[..., 3] > 0).float().unsqueeze(1)
depth = mask * depth


image = None
if feat is not None:
    image, _ = dr.interpolate(feat, rast_out, tri)
    image = image.permute(0, 3, 1, 2)
    image = mask * image

return mask, depth, image
```

IMG_8329	12/22/2024 12:27 PM	OBJ File	3,737 KB
IMG_8330	12/22/2024 12:27 PM	OBJ File	3,728 KB
IMG_8331	12/22/2024 12:27 PM	OBJ File	3,757 KB
IMG_8332	12/22/2024 12:27 PM	OBJ File	3,745 KB
IMG_8340	12/22/2024 12:27 PM	OBJ File	3,725 KB
IMG_8341	12/22/2024 12:27 PM	OBJ File	3,744 KB
IMG_8342	12/22/2024 12:27 PM	OBJ File	3,740 KB
IMG_8343	12/22/2024 12:27 PM	OBJ File	3,735 KB
IMG_8344	12/22/2024 12:27 PM	OBJ File	3,735 KB
IMG_8345	12/22/2024 12:27 PM	OBJ File	3,729 KB
IMG_8346	12/22/2024 12:27 PM	OBJ File	3,734 KB
IMG_8354	12/22/2024 12:27 PM	OBJ File	3,723 KB
IMG_8355	12/22/2024 12:27 PM	OBJ File	3,724 KB
IMG_8356	12/22/2024 12:27 PM	OBJ File	3,725 KB
IMG_8365	12/22/2024 12:27 PM	OBJ File	3,726 KB
IMG_8366	12/22/2024 12:27 PM	OBJ File	3,724 KB
IMG_8367	12/22/2024 12:27 PM	OBJ File	3,723 KB
IMG_8368	12/22/2024 12:27 PM	OBJ File	3,719 KB
IMG_8373	12/22/2024 12:27 PM	OBJ File	3,743 KB
IMG_8374	12/22/2024 12:27 PM	OBJ File	3,742 KB
IMG_8375	12/22/2024 12:27 PM	OBJ File	3,755 KB
IMG_8376	12/22/2024 12:27 PM	OBJ File	3,756 KB
IMG_8377	12/22/2024 12:27 PM	OBJ File	3,749 KB

rendering

- Use the extracted landmarks and textures to visualize the face.



libraries, computer specifications

❖ libraries

- Core Libraries:
  - Python: 3.8
  - OpenCV: For image processing tasks like loading, saving, and warping images.
  - NumPy: For numerical operations and data handling.
  - Matplotlib: For visualizing results during debugging or evaluation.
- Specialized Libraries:
  - Trimesh: For handling 3D geometry (e.g., saving reconstructed 3D meshes).
  - SciPy: For mathematical operations such as saving model coefficients as .mat files.
  - nvdiffrast: NVIDIA's differentiable rasterizer for rendering 3D meshes.
- Face Reconstruction:
  - Delaunay Triangulation (via OpenCV): For warping textures during texture blending.
- Data Preparation Utilities:
  - MeshLab: For rendering 3D meshes.

❖ **computer specifications**

Recommended Version/Specification		
CUDA Toolkit	11.8	
GPU	NVIDIA RTX 30xx or better, CUDA-enabled	
RAM	Recommended: 32 GB; Minimum: 16 GB	
Storage	SSD (1 TB or more)	
Operating System	Linux (Ubuntu 20.04/22.04) or Windows 10/11	

Thank you

Contact Info:
Yoon Joohyun
Dept. of Electrical and Computer Engineering
College of Engineering
INHA UNIVERSITY
Incheon, S. Korea
Tel : +82-(0)32-860-7406
E-mail: nayajoohyun@gmail.com
Web-page: autonav.inha.ac.kr