4. IMPLEMENTASI SISTEM

Pada bab ini akan membahas implementasi dari desain sistem yang telah dijelaskan pada bab sebelumnya.

Tabel 4.1 Hubungan Segmen Program dengan Desain Sistem

Segmen Program	Sub Bab Desain Sistem	Keterangan
Segmen Program 4.1	3.2.1	Konversi *Dataset*
Segmen Program 4.2	3.2.2.1	*Rendering Image*
Segmen Program 4.3	3.2.2.2	*Cropping Image*
Segmen Program 4.4	3.2.2.3	*Resizing* dan *Padding Image*
Segmen Program 4.5	3.2.2.4	*Rendering Depth Map*
Segmen Program 4.6	3.2.2.5	*Cropping Depth Map*
Segmen Program 4.7	3.2.2.6	*Resizing* dan *Padding Depth Map*
Segmen Program 4.8	3.2.2.7	*Parameter Tuning* dan Pembuatan *Mask*
Segmen Program 4.9	3.2.2.8	Pembuatan *Ground Truth Point Cloud*
Segmen Program 4.10	3.2.3	Model Generasi *Point Cloud*
Segmen Program 4.11	3.2.4	*Post Processing Fusion*

4.1 Konversi *Dataset*

Berikut adalah segmen program yang berhubungan dengan melakukan konversi dataset dari format OBJ menjadi GLB.

Segmen Program 4.1 Konversi *Dataset*

```
import os
import sys
import json
import shutil
import subprocess
from tqdm.auto import tqdm
taxonomy_list = json.loads(open("shapenetcore.taxonomy.json").read())
folder_dict = {}
for taxon in taxonomy_list:
    folder_dict[taxon["metadata"]["name"]] =
taxon["metadata"]["label"].split(",")[0]

    if "children" in taxon:
        for taxon_child in taxon["children"]:
```

```
            folder_dict[taxon_child["metadata"]["name"]] =
taxon_child["metadata"]["label"].split(",")[0]

            if "children" in taxon_child:
                for taxon_child_child in taxon_child["children"]:
                    folder_dict[taxon_child_child["metadata"]["name"]] =
taxon_child_child["metadata"]["label"].split(",")[0]

                    if "children" in taxon_child_child:
                        for taxon_child_child_child in
taxon_child_child["children"]:

folder_dict[taxon_child_child_child["metadata"]["name"]] =
taxon_child_child_child["metadata"]["label"].split(",")[0]

                            if "children" in taxon_child_child_child:
                                for taxon_child_child_child_child in
taxon_child_child_child["children"]:

folder_dict[taxon_child_child_child_child["metadata"]["name"]] =
taxon_child_child_child_child["metadata"]["label"].split(",")[0]

                                    if "children" in
taxon_child_child_child_child:
                                        for
taxon_child_child_child_child_child in
taxon_child_child_child_child["children"]:

folder_dict[taxon_child_child_child_child_child["metadata"]["name"]] =
taxon_child_child_child_child_child["metadata"]["label"].split(",")[0]
target_cat_path_list = []
obj_path_list = []
for folder_path in os.listdir("./ShapeNetCore"):

target_cat_path_list.append(f"./ShapeNetCoreGLB/{folder_dict[str(folder_path)]
}")
    for cat_path in os.listdir(f"./ShapeNetCore/{folder_path}"):
        source_obj_path =
os.path.abspath(f"./ShapeNetCore/{folder_path}/{cat_path}/models/model_normali
zed.obj")
        target_glb_path =
os.path.abspath(f"./ShapeNetCoreGLB/{folder_dict[str(folder_path)]}/{cat_path}
.glb")
        obj_path_list.append((source_obj_path, target_glb_path))
if os.path.exists("./ShapeNetCoreGLB/"):
    shutil.rmtree("./ShapeNetCoreGLB/")
os.mkdir("./ShapeNetCoreGLB/")
for new_dir in target_cat_path_list:
    os.mkdir(new_dir)
for curr_source, curr_target in tqdm(obj_path_list):
    subprocess.Popen(
```

```
        [
            'obj2gltf',
            '-i',
            f'{curr_source}',
            '-o',
            f'{curr_target}'
        ],
        shell=True,
        stderr=subprocess.STDOUT,
        universal_newlines=True,
    )
missing_list = []
for curr_source, curr_target in tqdm(obj_path_list):
    if not os.path.exists(curr_target):
        missing_list.append((curr_source, curr_target))
for curr_source, curr_target in tqdm(missing_list):
    if os.path.exists(curr_source):
        subprocess.check_output(
            [
                'obj2gltf',
                '-i',
                f'{curr_source}',
                '-o',
                f'{curr_target}'
            ],
            shell=True,
            stderr=subprocess.STDOUT,
            universal_newlines=True,
        )
missing_list_extended = []
for curr_source, curr_target in tqdm(obj_path_list):
    if not os.path.exists(curr_target):
        missing_list_extended.append((curr_source, curr_target))
```

4.2 Preprocessing

Berikut adalah segmen-segmen program yang berhubungan dengan *preprocessing dataset* sebelum menjadi input model.

Segmen Program 4.2 *Rendering Image*

```
import pyrender
import os
import trimesh
import numpy as np
import matplotlib.pyplot as plt
import numpy as np
import scipy
```

```
from scipy.io import savemat
from scipy.spatial.transform import Rotation
from sklearn.utils import check_random_state
from tqdm import tqdm
from trimesh.transformations import rotation_matrix, quaternion_from_matrix,
quaternion_matrix
import cv2
import PIL
from transforms3d.quaternions import quat2mat

rotation_angle_x = np.deg2rad(i * ROTATION_STEP)
rotation_matrix_x = Rotation.from_rotvec(rotation_angle_x * np.array([0, 1,
0])).as_matrix()

diagonal_angle_y = np.deg2rad(DIAGONAL_ANGLE)
diagonal_matrix_y = Rotation.from_rotvec(diagonal_angle_y * np.array([0, 1,
0])).as_matrix()

top_down_angle_x = np.deg2rad(TOP_DOWN_ANGLE)
top_down_matrix_x = Rotation.from_rotvec(top_down_angle_x * np.array([1, 0,
0])).as_matrix()

combined_rotation_matrix = np.dot(top_down_matrix_x, np.dot(diagonal_matrix_y
rotation_matrix_x))

# Set the object's pose with combined rotation
object_pose = np.eye(4)
object_pose[:3, :3] = combined_rotation_matrix
object_pose[:3, 3] = object_position

scene = pyrender.Scene()
# Create a camera
camera = pyrender.OrthographicCamera(xmag=0.5, ymag=0.5, znear=0.5, zfar=1.3)
camera_pose = np.eye(4)
camera_pose[:3, 3] = [0, 0, 1]
# Camera pose is not important for this example
scene.add(camera, pose=camera_pose)

# Create a light
```

```
light = pyrender.DirectionalLight(color=np.ones(3), intensity=2.0)
# Light pose is not important for this example
scene.add(light, pose=np.eye(4))

# Add the rotated mesh to the scene
scene.add(mesh, pose=object_pose)

# Render the scene
renderer_input = pyrender.OffscreenRenderer(
    viewport_width=512, viewport_height=512)
color_input, depth_input = renderer_input.render(scene)

# Explicitly release resources associated with the mesh
renderer_input.delete()
```

Segmen Program 4.3 *Cropping Image*

```
y_input_image = cv2.cvtColor(color_input, cv2.COLOR_BGR2GRAY)

# To find upper threshold, we need to apply Otsu's thresholding
upper_threshold, thresh_input_image = cv2.threshold(
gray_input_image,    thresh=0,    maxval=255,    type=cv2.THRESH_BINARY    +
cv2.THRESH_OTSU)

# Calculate lower threshold
lower_threshold = 0.5 * upper_threshold

# Apply canny edge detection
canny = cv2.Canny(color_input, lower_threshold, upper_threshold)
# Finding the non-zero points of canny
pts = np.argwhere(canny > 0)

# Finding the min and max points
try:
    y1, x1 = pts.min(axis=0)
    y2, x2 = pts.max(axis=0)

    # Crop ROI from the givn image
    output_image = color_input[y1:y2, x1:x2]
```

Segmen Program 4.4 *Resizing* dan *Padding Image*

```
old_size = output_image.shape[:2]
desired_img_size = 64

ratio_img = float(desired_img_size)/max(old_size)
new_img_size = tuple([int(x * ratio_img) for x in old_size])

im = cv2.resize(output_image, (new_img_size[1], new_img_size[0]))

delta_w_img = desired_img_size - new_img_size[1]
delta_h_img = desired_img_size - new_img_size[0]
top_img, bottom_img = delta_h_img//2, delta_h_img-(delta_h_img//2)
left_img, right_img = delta_w_img//2, delta_w_img-(delta_w_img//2)

new_im = cv2.copyMakeBorder(im, top_img, bottom_img, left_img, right_img,
    cv2.BORDER_CONSTANT,
    value=[255, 255, 255])
```

Segmen Program 4.5 *Rendering Depth Map*

```
quat_matrix = trans_file[j]
print(trans_file[j])
quat2rot_matrix = quat2mat(quat_matrix)

rotation_angle_z = np.deg2rad(-90)
rotation_axis_z = [0, 0, 1]
rotation_matrix_z = Rotation.from_rotvec(rotation_angle_z *
np.array(rotation_axis_z)).as_matrix()

rotation_angle_x = np.deg2rad(-90)
rotation_axis_x = [0, 1, 0]
rotation_matrix_x = Rotation.from_rotvec(rotation_angle_x *
np.array(rotation_axis_x)).as_matrix()

# Combine the rotation matrices
```

```
combined_rotation_matrix = np.dot(quat2rot_matrix, np.dot(rotation_matrix_x,
rotation_matrix_z))

object_pose = np.eye(4)
object_pose[:3, :3] = combined_rotation_matrix[:3, :3]

scene = pyrender.Scene()
# Create a camera
camera = pyrender.OrthographicCamera(xmag=0.5, ymag=0.5, znear=0.1, zfar=1.3)

camera_pose = np.eye(4)
camera_pose[:3, 3] = [0, 0, 1]
scene.add(camera, pose=camera_pose)

# Create a light
light = pyrender.DirectionalLight(color=np.ones(3), intensity=2.0)
scene.add(light, pose=np.eye(4))

# Add the rotated mesh to the scene
scene.add(mesh, pose=object_pose)

# Render the scene
renderer_input = pyrender.OffscreenRenderer(
viewport_width=512, viewport_height=512)
color_input, depth_input = renderer_input.render(scene)

# Explicitly release resources associated with the mesh
renderer_input.delete()
```

Segmen Program 4.6 *Cropping Depth Map*

```
gray_input_image = cv2.cvtColor(color_input, cv2.COLOR_BGR2GRAY)

# To find upper threshold, we need to apply Otsu's thresholding
upper_threshold, thresh_input_image = cv2.threshold(
gray_input_image, thresh=0, maxval=255, type=cv2.THRESH_BINARY + cv2.THRESH_OTSU
)
# Calculate lower threshold
lower_threshold = 0.5 * upper_threshold
```

```
# Apply canny edge detection
canny = cv2.Canny(color_input, lower_threshold, upper_threshold)

# Finding the non-zero points of canny
pts = np.argwhere(canny > 0)

# Finding the min and max points
try:
    y1, x1 = pts.min(axis=0)
    y2, x2 = pts.max(axis=0)

    # Crop ROI from the given image
    output_depth = depth_input[y1:y2, x1:x2]
```

Segmen Program 4.7 *Resizing* dan *Padding Depth Map*

```
old_size = output_depth.shape[:2]
desired_dp_size = 128

ratio_dp = float(desired_dp_size)/max(old_size)
new_dp_size = tuple([int(x * ratio_dp) for x in old_size])

dp = cv2.resize(output_depth, (new_dp_size[1], new_dp_size[0]))

delta_w_dp = desired_dp_size - new_dp_size[1]
delta_h_dp = desired_dp_size - new_dp_size[0]
top_dp, bottom_dp = delta_h_dp//2, delta_h_dp-(delta_h_dp//2)
left_dp, right_dp = delta_w_dp//2, delta_w_dp-(delta_w_dp//2)

new_depth = cv2.copyMakeBorder(dp, top_dp, bottom_dp, left_dp, right_dp,
cv2.BORDER_CONSTANT, value=[0, 0, 0])
```

Segmen Program 4.8 *Parameter Tuning* dan Pembuatan *Mask*

```
depth = np.load(
            f"{self.cfg.path}/depth/{classname}/{filename}.npy")
```

```
mask = np.array([value != 0 for value in depth])
mask = mask.astype(bool)
depth = depth.transpose(1, 2, 0)
depth = (depth - np.min(depth, axis=(0, 1))) / \
        (np.max(depth, axis=(0, 1)) - np.min(depth, axis=(0, 1)))
depth = depth.transpose(2, 0, 1)

depth[depth == 0] = 0.5

background_mask = (depth == 0)
min_val = np.min(depth[depth != 0])
max_val = np.max(depth[depth != 0])

# Generate random noise values for the background pixels
noise_range = (min_val, max_val)  # Adjust the range as needed
noise_values = np.random.uniform(
noise_range[0], noise_range[1], size=depth.shape)

# Replace the background pixels with the generated noise values
depth_with_noise = depth.copy()
depth_with_noise[background_mask] = noise_values[background_mask]
```

Segmen Program 4.9 Pembuatan *Ground Truth Point Cloud*

```
def create_dense_point_cloud(vertices, faces, density=1000, random_state=None):
    """
    Create a dense point cloud by subdividing faces and sampling points within
them.

    Parameters:
        vertices (numpy.ndarray): 2D array of vertex coordinates (n_vertices x
3).
        faces (numpy.ndarray): 2D array of faces (n_faces x 3).
        density (int): Number of points to sample per unit area.
        random_state (int or RandomState, optional): Seed or RandomState for
reproducibility.

    Returns:
        numpy.ndarray: Dense point cloud coordinates (n_points x 3).
```

```
    """
    # Ensure vertices and faces are NumPy arrays
    vertices = np.array(vertices)
    faces = np.array(faces)

    # Subdivide faces and sample points within each triangle
    random_state = check_random_state(random_state)
    mesh = trimesh.Trimesh(vertices=vertices, faces=faces)

    # Calculate the total area of the triangles in the mesh
    total_area = np.sum(trimesh.triangles.area(mesh.triangles))

    # Calculate the number of points to sample
    num_points = int(density * total_area)

    dense_point_cloud, _ = trimesh.sample.sample_surface(
        mesh,
        num_points,
        seed=random_state.randint(2**31)
    )

    return dense_point_cloud

def visualize_point_cloud(point_cloud):
    """
    Visualize a point cloud using matplotlib.

    Parameters:
        point_cloud (numpy.ndarray): Dense point cloud coordinates (n_points x
3).
    """
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')

    # Extract x, y, and z coordinates from the point cloud
    x = point_cloud[:, 0]
    y = point_cloud[:, 1]
    z = point_cloud[:, 2]
```

```
    ax.scatter(x, y, z, s=1, c='b', marker='o')  # Adjust marker size (s) as
needed

    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    plt.show()
```

4.3 Model Generasi *Point Cloud* dan Implementasi *Fusing*

Berikut adalah segmen-segmen program yang berkaitan dengan arsitektur model serta proses *post processing*, yaitu implementasi *fusing* dengan *invers matriks*.

Segmen Program 4.10 Model Generasi *Point Cloud*

```
import torch
from torch import nn
from torch.nn import functional as F
import config


def conv2d_block(in_c, out_c):
    return nn.Sequential(
        nn.Conv2d(in_c, out_c, 3, stride=2, padding=1),
        nn.BatchNorm2d(out_c),
        nn.ReLU(),
    )


def deconv2d_block(in_c, out_c):
    return nn.Sequential(
        nn.Conv2d(in_c, out_c, 3, stride=1, padding=1),
        nn.BatchNorm2d(out_c),
        nn.ReLU(),
    )


def linear_block(in_c, out_c):
```

```
    return nn.Sequential(
        nn.Linear(in_c, out_c),
        nn.BatchNorm1d(out_c),
        nn.ReLU(),
    )


def pixel_bias(inputViewN, outW, outH, renderDepth):
    X, Y = torch.meshgrid([torch.arange(outH), torch.arange(outW)])
    X, Y = X.float(), Y.float()  # [H,W]
    initTile = torch.cat([
        X.repeat([inputViewN, 1, 1]),  # [V,H,W]
        Y.repeat([inputViewN, 1, 1]),  # [V,H,W]
        torch.ones([inputViewN, outH, outW]).float() * renderDepth,
        torch.zeros([inputViewN, outH, outW]).float(),
    ], dim=0)  # [4V,H,W]

    return initTile.unsqueeze_(dim=0)  # [1,4V,H,W]


class Encoder(nn.Module):
    """Encoder of Structure Generator"""

    def __init__(self, inputViewN):
        super(Encoder, self).__init__()

        self.conv1 = conv2d_block(3*inputViewN, 96)
        self.conv2 = conv2d_block(96, 128)
        self.conv3 = conv2d_block(128, 192)
        self.conv4 = conv2d_block(192, 256)
        self.fc1 = linear_block(4096, 2048)  # After flatten
        self.fc2 = linear_block(2048, 1024)
        self.fc3 = nn.Linear(1024, 512)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = self.conv4(x)
```

```
        x = self.fc1(x.reshape(-1, 4096))
        x = self.fc2(x)
        x = self.fc3(x)

        return x


class Decoder(nn.Module):
    """Build Decoder"""

    def __init__(self, inputViewN, outW, outH, renderDepth):
        super(Decoder, self).__init__()
        self.inputViewN = inputViewN

        self.relu = nn.ReLU()
        self.fc1 = linear_block(512, 1024)
        self.fc2 = linear_block(1024, 2048)
        self.fc3 = linear_block(2048, 4096)
        self.deconv1 = deconv2d_block(256, 192)
        self.deconv2 = deconv2d_block(192, 128)
        self.deconv3 = deconv2d_block(128, 96)
        self.deconv4 = deconv2d_block(96, 64)
        self.deconv5 = deconv2d_block(64, 48)
        self.pixel_conv = nn.Conv2d(48, inputViewN*4, 1, stride=1, bias=False)
        self.pixel_bias = pixel_bias(inputViewN, outW, outH, renderDepth)

    def forward(self, x):
        x = self.relu(x)
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.fc3(x)
        x = x.reshape([-1, 256, 4, 4])
        x = self.deconv1(F.interpolate(x, scale_factor=2))
        x = self.deconv2(F.interpolate(x, scale_factor=2))
        x = self.deconv3(F.interpolate(x, scale_factor=2))
        x = self.deconv4(F.interpolate(x, scale_factor=2))
        x = self.deconv5(F.interpolate(x, scale_factor=2))
        x = self.pixel_conv(x) + self.pixel_bias.to(x.device)
        XYZ, maskLogit = torch.split(
```

```
            x, [self.inputViewN * 3, self.inputViewN], dim=1)

        return XYZ, maskLogit


class Generator(nn.Module):
    def __init__(self, encoder=None, decoder=None, inputViewN=4,
                 outW=128, outH=128, renderDepth=1.0):
        super(Generator, self).__init__()

        if encoder:
            self.encoder = encoder
        else:
            self.encoder = Encoder(inputViewN)

        if decoder:
            self.decoder = decoder
        else:
            self.decoder = Decoder(inputViewN, outW, outH, renderDepth)

    def forward(self, x):
        latent = self.encoder(x)
        XYZ, maskLogit = self.decoder(latent)

        return XYZ, maskLogit


class Discriminator(nn.Module):
    def __init__(self, inputViewN=8):
        super(Discriminator, self).__init__()
        self.main = nn.Sequential(
            # input is ``(nc) x 64 x 64``
            nn.Conv2d(inputViewN, 128, 3, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # nn.Dropout(p=0.2),

            # state size. ``(ndf) x 32 x 32``
            nn.Conv2d(128, 256 * 2, 3, 2, 1, bias=False),
            nn.BatchNorm2d(256 * 2),
```

```
            nn.LeakyReLU(0.2, inplace=True),
            # nn.Dropout(p=0.2),

            # state size. ``(ndf*2) x 16 x 16``
            nn.Conv2d(256 * 2, 512 * 4, 3, 2, 1, bias=False),
            nn.BatchNorm2d(512 * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # nn.Dropout(p=0.2),

            # state size. ``(ndf*4) x 8 x 8``
            nn.Conv2d(512 * 4, 1024 * 8, 3, 2, 1, bias=False),
            # nn.BatchNorm2d(1024 * 8),
            # nn.LeakyReLU(0.2, inplace=True),

            # state size. ``(ndf*8) x 4 x 4``
            # nn.Conv2d(1024 * 8, 1, 3, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        return self.main(input)


if __name__ == '__main__':
    import config
    cfg = config.get_arguments()
    encoder = Encoder(cfg.inputViewN)
    decoder = Decoder(cfg.inputViewN, cfg.outW, cfg.outH, cfg.renderDepth)

    generator = Generator()
    discriminator = Discriminator()
```

Segmen Program 4.11 *Post Processing Fusion*

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
```

```
from trimesh.transformations import quaternion_from_matrix, quaternion_matrix,
inverse_matrix, translation_matrix, rotation_matrix


def fuse3D(cfg, XYZ, maskLogit, fuseTrans, chunkSize):
    """Fuse multiple depth views into a 3D point cloud representation
    Args:
    output of structure generator
        XYZ (tensor:[B,3V,H,W]): x,z,y of V different fixed views
        maskLogit (tensor:[B,V,H,W]): mask of V different fixed views
    output of render module
        fuseTrans (Tensor:[V, 4])
    Return:
        XYZid (Tensor [B,3,VHW]): point clouds
        ML (Tensor [B,1,VHW]): depth stack
     """
    t_view = torch.Tensor([1, 1, 0])

    # t_view = torch.Tensor([1, 1, 0])

    # t_view = torch.Tensor([1, 0, 0])

    transMat = translation_matrix(t_view)
    transMat = transMat[None, ...]  # expand to 1,4,4
    transMat = np.repeat(transMat, cfg.inputViewN,
                         axis=0)  # repeat 1 to n views
    transMat = torch.from_numpy(transMat).float().to(cfg.device)

    q_view = fuseTrans  # [V, 4]

    rotMat3 = quaternionToRotMatrix(q_view)
    rotMat = torch.eye(4)[None, ...].repeat(
        [cfg.inputViewN, 1,  1]).to(cfg.device)
    rotMat[:, :3, :3] = rotMat3
    rotMat = rotMat.float().to(cfg.device)

    # 2D to 3D coordinate transformation
    khomMat = cfg.Khom2Dto3D  # [4x4]
    khomMat = khomMat.repeat(
```

```
        [cfg.inputViewN, 1, 1])  # [B,V,4x4]

    rotation_angle_post = np.deg2rad(90)  # orig
    rotation_axis_post = [0, 0, 1]

    rotationPostMat      =      np.repeat(rotation_matrix(rotation_angle_post,
rotation_axis_post)[None, ...], cfg.inputViewN,
                                axis=0)
    rotationPostMat = torch.from_numpy(rotationPostMat).float().to(cfg.device)

    inverseAllTransMat = torch.inverse(torch.matmul(
        khomMat,    torch.matmul(transMat,        torch.matmul(rotationPostMat,
rotMat))))

    # transform depth stack
    ML = maskLogit.clone().reshape([chunkSize, 1, -1])  # [B,1,VHW]
    XYZhom = get3DhomCoord(XYZ, cfg, chunkSize)  # [B,V,4,HW]
    XYZid = torch.matmul(inverseAllTransMat, XYZhom)  # [B,V,3,HW]
    XYZid = XYZid[:, :, :3, :]  # [B,V,3,HW]

    # fuse point clouds
    XYZid = XYZid.permute([0, 2, 1, 3]).reshape(
        [chunkSize, 3, -1])  # [B,3,VHW]

    return XYZid, ML

def quaternionToRotMatrix(q):
    # q = [V, 4]
    qa, qb, qc, qd = torch.unbind(q, dim=1)  # [V,]
    R = torch.stack(
        [torch.stack([1 - 2 * (qc**2 + qd**2),
                      2 * (qb * qc - qa * qd),
                      2 * (qa * qc + qb * qd)]),
         torch.stack([2 * (qb * qc + qa * qd),
                      1 - 2 * (qb**2 + qd**2),
                      2 * (qc * qd - qa * qb)]),
         torch.stack([2 * (qb * qd - qa * qc),
                      2 * (qa * qb + qc * qd),
                      1 - 2 * (qb**2 + qc**2)])]
```

```
    ).permute(2, 0, 1)
    return R.to(q.device)


def transParamsToHomMatrix(q, t):
    """q = [V, 4], t = [V,3]"""
    N = q.size(0)
    R = quaternionToRotMatrix(q)  # [V,3,3]
    Rt = torch.cat([R, t.unsqueeze(-1)], dim=2)  # [V,3,4]
    hom_aug = torch.cat([torch.zeros([N, 1, 3]), torch.ones([N, 1, 1])],
                        dim=2).to(Rt.device)
    RtHom = torch.cat([Rt, hom_aug], dim=1)  # [V,4,4]
    return RtHom


def get3DhomCoord(XYZ, cfg, chunkSize):
    ones = torch.ones([chunkSize, cfg.inputViewN, cfg.outH, cfg.outW]) \
        .to(XYZ.device)
    XYZhom = torch.cat([XYZ, ones], dim=1) \
        .reshape([chunkSize, 4, cfg.inputViewN, -1])\
        .permute([0, 2, 1, 3])

    return XYZhom  # [B,V,4,HW]
```