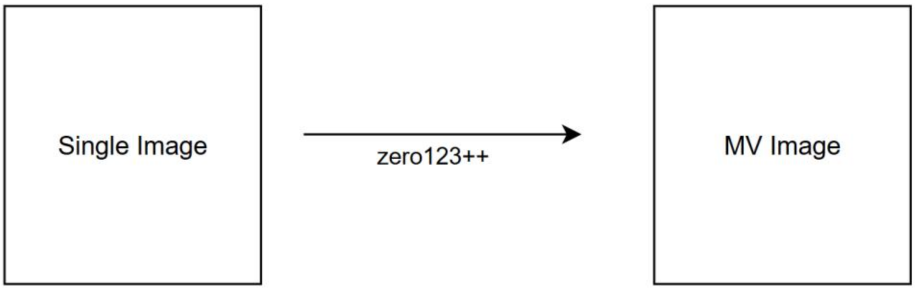
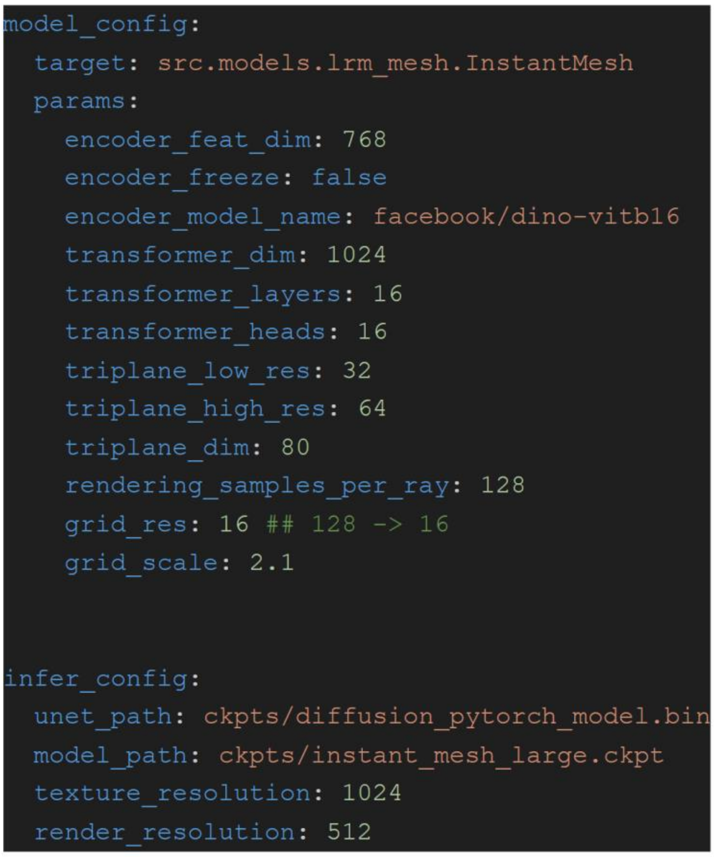




```
model_config:
  target: src.models.lrm_mesh.InstantMesh
  params:
    encoder_feat_dim: 768
    encoder_freeze: false
    encoder_model_name: facebook/dino-vitb16
    transformer_dim: 1024
    transformer_layers: 16
    transformer_heads: 16
    triplane_low_res: 32
    triplane_high_res: 64
    triplane_dim: 80
    rendering_samples_per_ray: 128
    grid_res: 16 ## 128 -> 16
    grid_scale: 2.1


infer_config:
  unet_path: ckpts/diffusion_pytorch_model.bin
  model_path: ckpts/instant_mesh_large.ckpt
  texture_resolution: 1024
  render_resolution: 512
```

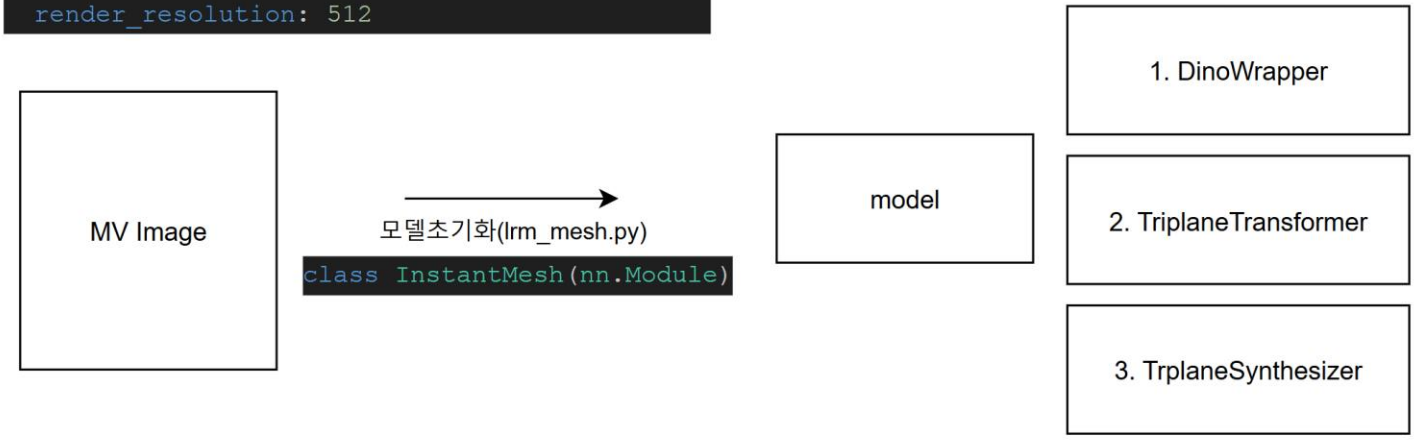
기존 LRM

이미지: 512 512
512 * 512 -> 32*32 개의 Patches (Conv)
-> 512 *512 * 768

instantmesh

320*320*6
zero123 -> 320*320 -> 20*20, 400 + 1(CLS)
-> 최종 401 * 768

이게6장이므로, 2401 * 768



```
class InstantMesh(nn.Module)
```

1. DinoWrapper

```
InstantMesh
(
  (encoder): DinoWrapper
  (
    (model): ViTModel
    (
      (embeddings): ViTEmbeddings
      (
        (patch_embeddings): ViTPatchEmbeddings
        (
          (projection): Conv2d(3, 768, kernel_size=(16, 16), stride=(16, 16))
        )
        (dropout): Dropout(p=0.0, inplace=False)
      )
      (encoder): ViTEncoder
      (
        (layer): ModuleList
        (
          (0-11): 12 x ViTLayer
          (
            (attention): ViTAttention
            (
              (attention): ViTSelfAttention
              (
                (query): Linear(in_features=768, out_features=768, bias=True)
                (key): Linear(in_features=768, out_features=768, bias=True)
                (value): Linear(in_features=768, out_features=768, bias=True)
                (dropout): Dropout(p=0.0, inplace=False)
              )
              (output): ViTSelfOutput
              (
                (dense): Linear(in_features=768, out_features=768, bias=True)
                (dropout): Dropout(p=0.0, inplace=False)
              )
            )
            (intermediate): ViTIntermediate
            (
              (dense): Linear(in_features=768, out_features=3072, bias=True)
              (intermediate_act_fn): GELUActivation()
            )
            (output): ViTOutput
            (
              (dense): Linear(in_features=3072, out_features=768, bias=True)
              (dropout): Dropout(p=0.0, inplace=False)
            )
            (layernorm_before): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (layernorm_after): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (adaLN_modulation): Sequential
            (
              (0): SiLU()
              (1): Linear(in_features=768, out_features=3072, bias=True)
            )
          )
        )
      )
      (layernorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
    )


    (camera_embedder): Sequential
    (
      (0): Linear(in_features=16, out_features=768, bias=True)
      (1): SiLU()
      (2): Linear(in_features=768, out_features=768, bias=True)
    )

  )
)
```

2. TriplaneTransformer

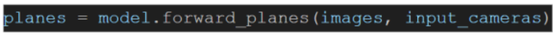
```
(transformer): TriplaneTransformer
 (
  (layers): ModuleList
  (
   (0-15): 16 x BasicTransformerBlock
   (
    (norm1): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
    (cross_attn): MultiheadAttention
    (
     (out_proj): NonDynamicallyQuantizableLinear(in_features=1024, out_features=1024, bias=False)
    )
    (norm2): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
    (self_attn): MultiheadAttention
    (
     (out_proj): NonDynamicallyQuantizableLinear(in_features=1024, out_features=1024, bias=False)
    )
    (norm3): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
    (mlp): Sequential
    (
     (0): Linear(in_features=1024, out_features=4096, bias=True)
     (1): GELU(approximate='none')
     (2): Dropout(p=0.0, inplace=False)
     (3): Linear(in_features=4096, out_features=1024, bias=True)
     (4): Dropout(p=0.0, inplace=False)
    )
   )
  )
  (norm): LayerNorm((1024,), eps=1e-06, elementwise_affine=True)
  (deconv): ConvTranspose2d(1024, 80, kernel_size=(2, 2), stride=(2, 2))

 )
```

3. Synthesizer

```
(synthesizer): TriplaneSynthesizer
  (
    (decoder): OSGDecoder
    (
      (net_sdf): Sequential
      (
        (0): Linear(in_features=240, out_features=64, bias=True)
        (1): ReLU()
        (2): Linear(in_features=64, out_features=64, bias=True)
        (3): ReLU()
        (4): Linear(in_features=64, out_features=64, bias=True)
        (5): ReLU()
        (6): Linear(in_features=64, out_features=1, bias=True)
      )
      (net_rgb): Sequential
      (
        (0): Linear(in_features=240, out_features=64, bias=True)
        (1): ReLU()
        (2): Linear(in_features=64, out_features=64, bias=True)
        (3): ReLU()
        (4): Linear(in_features=64, out_features=64, bias=True)
        (5): ReLU()
        (6): Linear(in_features=64, out_features=3, bias=True)
      )
      (net_deformation): Sequential
      (
        (0): Linear(in_features=240, out_features=64, bias=True)
        (1): ReLU()
        (2): Linear(in_features=64, out_features=64, bias=True)
        (3): ReLU()
        (4): Linear(in_features=64, out_features=64, bias=True)
        (5): ReLU()
        (6): Linear(in_features=64, out_features=3, bias=True)
      )
      (net_weight): Sequential
      (
        (0): Linear(in_features=1920, out_features=64, bias=True)
        (1): ReLU()
        (2): Linear(in_features=64, out_features=64, bias=True)
        (3): ReLU()
        (4): Linear(in_features=64, out_features=64, bias=True)
        (5): ReLU()
        (6): Linear(in_features=64, out_features=21, bias=True)
      )
    )
  )
)
```

이미지처리순서
1.encoder

1. ft.py

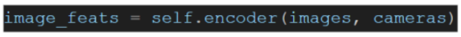


```
planes = model.forward_planes(images, input_cameras)
```

images.shape
torch.Size([1, 6, 3, 320, 320])

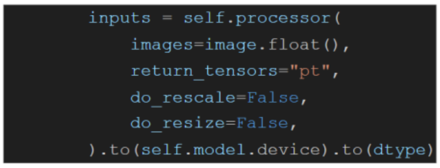
cameras.shape
torch.Size([1, 6, 16])

2. ft.py -> lrm_mesh.py



```
image_feats = self.encoder(images, cameras)
```

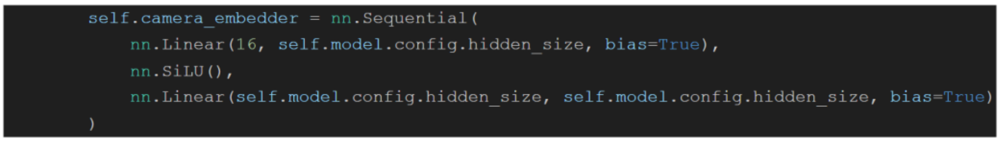
3. ft.py -> lrm_mesh.py -> dino_wrapper.py
(이미지를 inputs로)



```
        inputs = self.processor(
            images=image.float(),
            return_tensors="pt",
            do_rescale=False,
            do_resize=False,
        ).to(self.model.device).to(dtype)
```

inputs['pixel_values'].shape
torch.Size([6, 3, 320, 320])

3-1. ft.py -> lrm_mesh.py -> dino_wrapper.py
(카메라 임베딩)

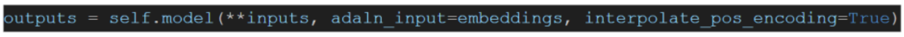


```
        self.camera_embedder = nn.Sequential(
            nn.Linear(16, self.model.config.hidden_size, bias=True),
            nn.SiLU(),
            nn.Linear(self.model.config.hidden_size, self.model.config.hidden_size, bias=True)
        )
```

cameras.shape
torch.Size([1, 6, 16])

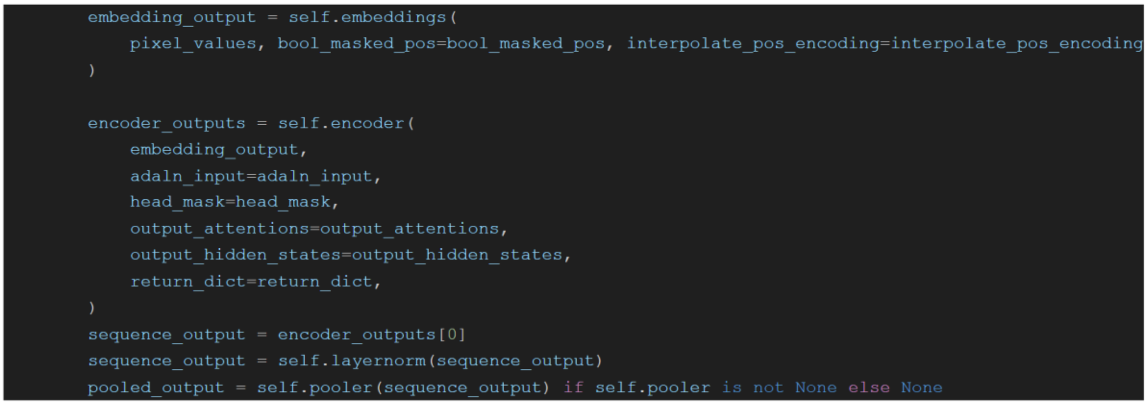
아웃풋 -> [1 6 768]

4. ft.py -> lrm_mesh.py -> dino_wrapper.py
전처리된 이미지(inputs), 카메라(embeddings) 로 인코더 태움



```
outputs = self.model(**inputs, adaln_input=embeddings, interpolate_pos_encoding=True)
```

5. ft.py -> lrm_mesh.py -> dino_wrapper.py -> dino.py
ViTModel 의 forward 메소드 호출



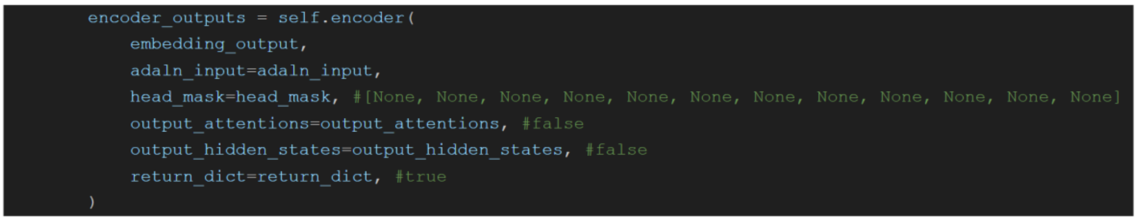
```
        embedding_output = self.embeddings(
            pixel_values, bool_masked_pos=bool_masked_pos, interpolate_pos_encoding=interpolate_pos_encoding
        )

        encoder_outputs = self.encoder(
            embedding_output,
            adaln_input=adaln_input,
            head_mask=head_mask,
            output_attentions=output_attentions,
            output_hidden_states=output_hidden_states,
            return_dict=return_dict,
        )
        sequence_output = encoder_outputs[0]
        sequence_output = self.layernorm(sequence_output)
        pooled_output = self.pooler(sequence_output) if self.pooler is not None else None
```

5-1. ft.py -> lrm_mesh.py -> dino_wrapper.py -> dino.py
embedding 수행, Conv2d(3, 768, kernel_size=(16, 16), stride=(16, 16))
인풋은 6 320 320 3 (인풋채널은 RGB)

1. 320*320 이미지를 16*16 패치로 분할 -> 20*20 개의 패치 생성
2. 각 패치를 768차원 벡터로 변환
3. 최종 임베딩결과 400*768 + cls token 1 * 768 -> 401*768 (랜덤하게 init)

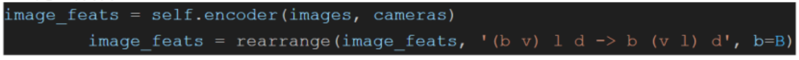
embedding_output.shape
torch.Size([6, 401, 768])

5-2. ft.py -> lrm_mesh.py -> dino_wrapper.py -> dino.py
embedding 수행후 encoder 태우기



```
        encoder_outputs = self.encoder(
            embedding_output,
            adaln_input=adaln_input,
            head_mask=head_mask, #[None, None, None, None, None, None, None, None, None, None, None, None]
            output_attentions=output_attentions, #false
            output_hidden_states=output_hidden_states, #false
            return_dict=return_dict, #true
        )
```

6. 최종 아웃풋

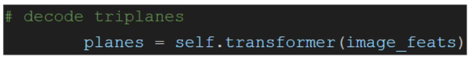


```
image_feats = self.encoder(images, cameras)
        image_feats = rearrange(image_feats, '(b v) l d -> b (v l) d', b=B)
```

torch.Size([1, 2406, 768])

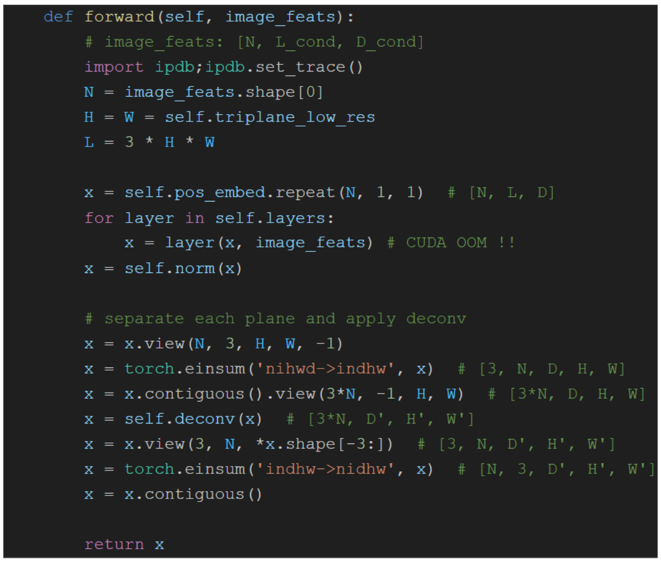
2. triplane transformer(Decoder)

1. ft.py -> lrm_mesh.py

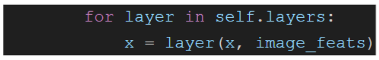


```
# decode triplanes
        planes = self.transformer(image_feats)
```

2. ft.py -> lrm_mesh.py -> decoder/transformer.py

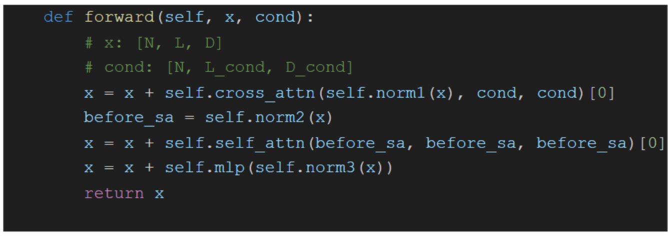


```
    def forward(self, image_feats):
        # image_feats: [N, L_cond, D_cond]
        import ipdb;ipdb.set_trace()
        N = image_feats.shape[0]
        H = W = self.triplane_low_res
        L = 3 * H * W

        x = self.pos_embed.repeat(N, 1, 1)  # [N, L, D]
        for layer in self.layers:
            x = layer(x, image_feats) # CUDA OOM !!
        x = self.norm(x)

        # separate each plane and apply deconv
        x = x.view(N, 3, H, W, -1)
        x = torch.einsum('nihwd->indhw', x)  # [3, N, D, H, W]
        x = x.contiguous().view(3*N, -1, H, W)  # [3*N, D, H, W]
        x = self.deconv(x)  # [3*N, D', H', W']
        x = x.view(3, N, *x.shape[-3:])  # [3, N, D', H', W']
        x = torch.einsum('indhw->nidhw', x)  # [N, 3, D', H', W']
        x = x.contiguous()

        return x
```



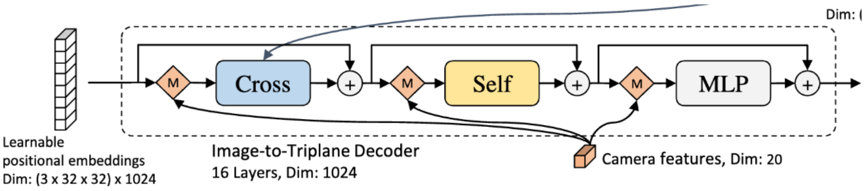
```
        for layer in self.layers:
            x = layer(x, image_feats)
```

3. ft.py -> lrm_mesh.py , 16 layers
x : 3072 * 1024 , cond : 2406 * 768

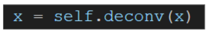


```
    def forward(self, x, cond):
        # x: [N, L, D]
        # cond: [N, L_cond, D_cond]
        x = x + self.cross_attn(self.norm1(x), cond, cond)[0]
        before_sa = self.norm2(x)
        x = x + self.self_attn(before_sa, before_sa, before_sa)[0]
        x = x + self.mlp(self.norm3(x))
        return x
```

파인튜닝시 cuda 아끼기 -> encoding 은 한번만 수행함 + decoder forward 에서 메모리 아끼기



4. ft.py -> lrm_mesh.py , 16 layers
이후 deconv -> triplane feature 3*64*64*80



```
x = self.deconv(x)
```

3. forward_geometry 로 image 구하기

```
    def forward_geometry(self, planes, render_cameras, render_size=256):
        '''
        Main function of our Generator. It first generate 3D mesh, then render it into 2D image
        with given `render_cameras`.
        :param planes: triplane features
        :param render_cameras: cameras to render generated 3D shape
        '''
```

1. `get_geometry_prediction`

```
 # Generate 3D mesh first
        mesh_v, mesh_f, sdf, deformation, v_deformed, sdf_reg_loss = self.get_geometry_prediction(planes)
```

1-1. `get_sdf_deformation_prediction`

```
# Step 1: first get the sdf and deformation value for each vertices in the tetrahedon grid.
        sdf, deformation, sdf_reg_loss, weight = self.get_sdf_deformation_prediction(planes)
```

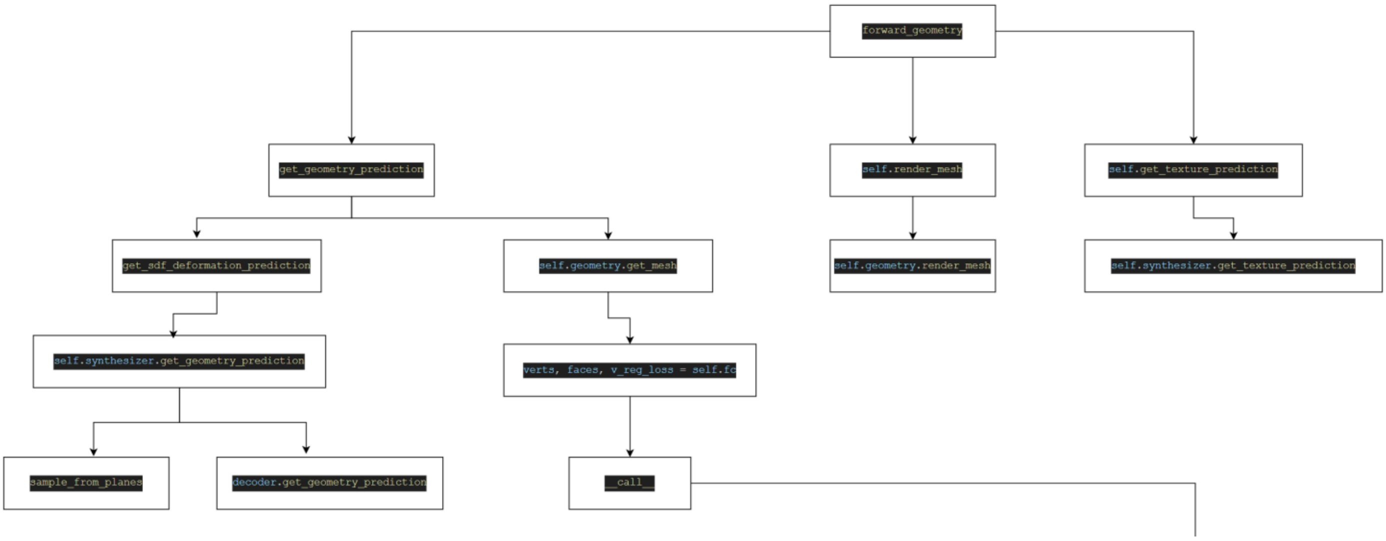
1-1-1. 예시 grid_res = 4 (easy)

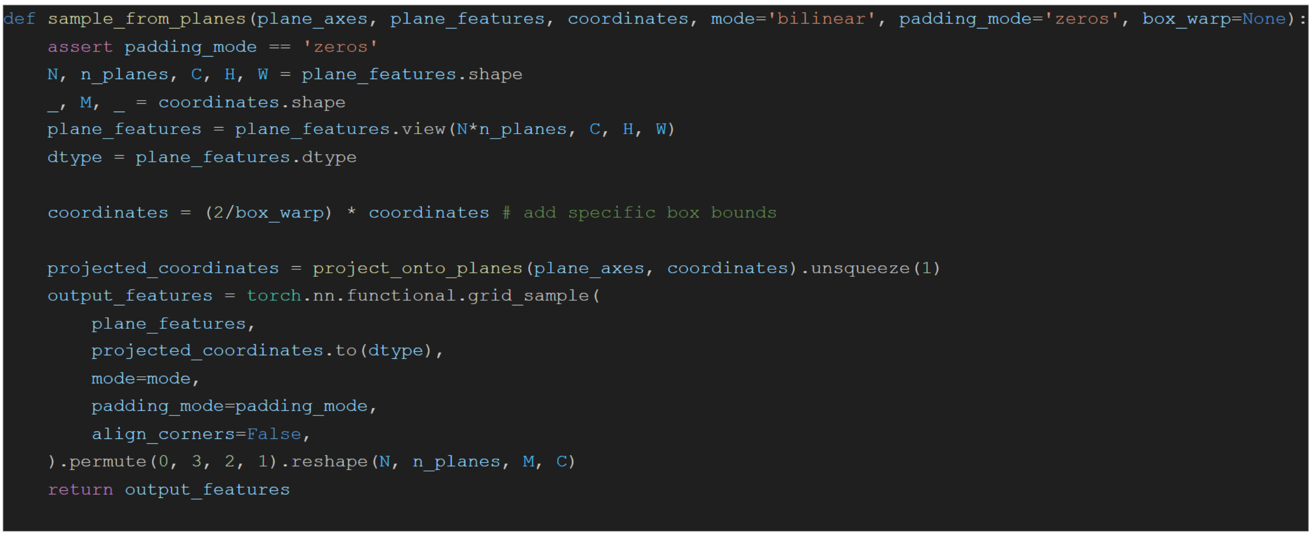
```
    def get_sdf_deformation_prediction(self, planes):
        '''
        Predict SDF and deformation for tetrahedron vertices
        :param planes: triplane feature map for the geometry
        '''
        init_position = self.geometry.verts.unsqueeze(0).expand(planes.shape[0], -1, -1)
```

```
        sdf, deformation, weight = torch.utils.checkpoint.checkpoint(
            self.synthesizer.get_geometry_prediction,
            planes,
            init_position,
            self.geometry.indices,
            use_reentrant=False,
        )
```

> /home/jinho99/anaconda3/envs/im/lib/python3.10/site-packages/torch/utils/checkpoint.py(342)checkpoint()

```
    def get_geometry_prediction(self, planes, sample_coordinates, flexicubes_indices):
        plane_axes = self.plane_axes.to(planes.device)
        sampled_features = sample_from_planes(
            plane_axes, planes, sample_coordinates, padding_mode='zeros', box_warp=self.rendering_kwargs['box_warp'])

        sdf, deformation, weight = self.decoder.get_geometry_prediction(sampled_features, flexicubes_indices)
        return sdf, deformation, weight
```


forward_geometry
get_geometry_prediction
self.render_mesh
self.get_texture_prediction
get_sdf_deformation_prediction
self.geometry.get_mesh
self.geometry.render_mesh
self.synthesizer.get_texture_prediction
self.synthesizer.get_geometry_prediction
verts, faces, v_reg_loss = self.fc
sample_from_planes
decoder.get_geometry_prediction
__call__

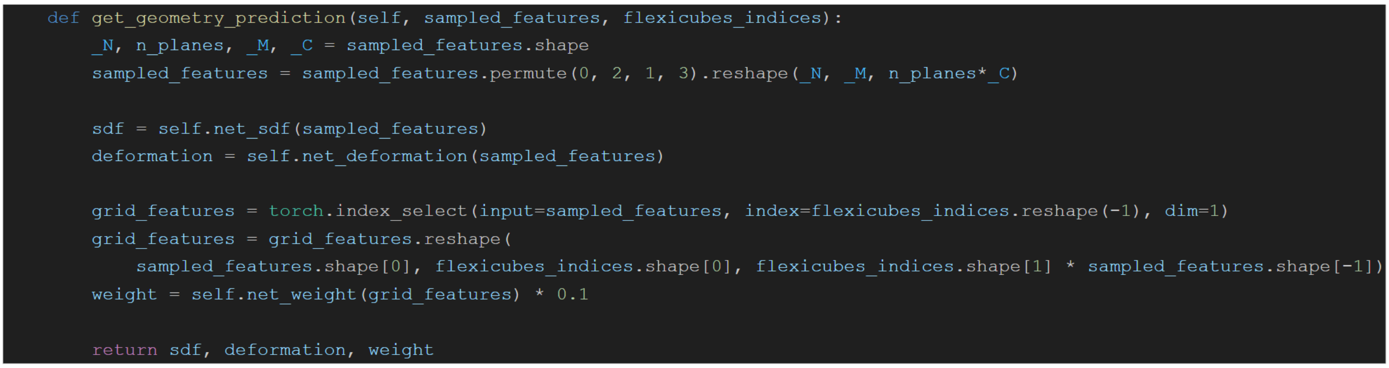


```
def sample_from_planes(plane_axes, plane_features, coordinates, mode='bilinear', padding_mode='zeros', box_warp=None):
    assert padding_mode == 'zeros'
    N, n_planes, C, H, W = plane_features.shape
    _, M, _ = coordinates.shape
    plane_features = plane_features.view(N*n_planes, C, H, W)
    dtype = plane_features.dtype

    coordinates = (2/box_warp) * coordinates # add specific box bounds

    projected_coordinates = project_onto_planes(plane_axes, coordinates).unsqueeze(1)
    output_features = torch.nn.functional.grid_sample(
        plane_features,
        projected_coordinates.to(dtype),
        mode=mode,
        padding_mode=padding_mode,
        align_corners=False,
    ).permute(0, 3, 2, 1).reshape(N, n_planes, M, C)
    return output_features
```

```
ipdb> output_features.shape
torch.Size([1, 3, 125, 80]).     <- 125는 vertex 갯수
```

```
ipdb> projected_coordinates.shape
torch.Size([3, 1, 125, 2])
```

```
ipdb> coordinates.shape
torch.Size([1, 125, 3])
```



```
    def get_geometry_prediction(self, sampled_features, flexicubes_indices):
        _N, n_planes, _M, _C = sampled_features.shape
        sampled_features = sampled_features.permute(0, 2, 1, 3).reshape(_N, _M, n_planes*_C)

        sdf = self.net_sdf(sampled_features)
        deformation = self.net_deformation(sampled_features)

        grid_features = torch.index_select(input=sampled_features, index=flexicubes_indices.reshape(-1), dim=1)
        grid_features = grid_features.reshape(
            sampled_features.shape[0], flexicubes_indices.shape[0], flexicubes_indices.shape[1] * sampled_features.shape[-1])
        weight = self.net_weight(grid_features) * 0.1

        return sdf, deformation, weight
```

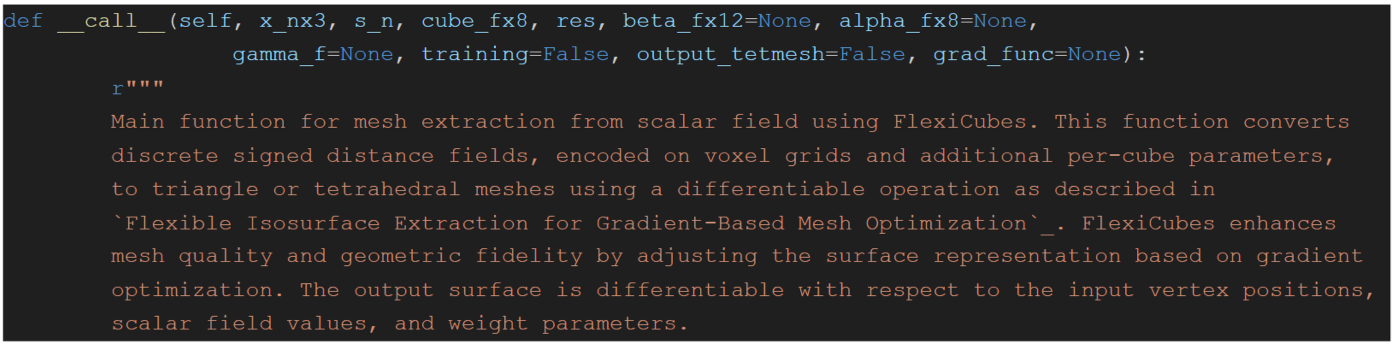
```
ipdb> sampled_features.shape
torch.Size([1, 3, 125, 80])
ipdb> flexicubes_indices.shape
torch.Size([64, 8])
ipdb>
```

sdf, deform : 240 (3*80) -> 1

```
ipdb> grid_features.shape
torch.Size([1, 512, 240])
```

```
ipdb> grid_features.shape
torch.Size([1, 64, 1920])
ipdb> self.net_weight
Sequential(
  (0): Linear(in_features=1920, out_features=64, bias=True)
  (1): ReLU()
  (2): Linear(in_features=64, out_features=64, bias=True)
  (3): ReLU()
  (4): Linear(in_features=64, out_features=64, bias=True)
  (5): ReLU()
  (6): Linear(in_features=64, out_features=21, bias=True)
)
```

```
ipdb> sdf.shape
torch.Size([1, 125, 1])
ipdb> deformation.shape
torch.Size([1, 125, 3])
ipdb> weight.shape
torch.Size([1, 64, 21])
```



```
def __call__(self, x_nx3, s_n, cube_fx8, res, beta_fx12=None, alpha_fx8=None,
                 gamma_f=None, training=False, output_tetmesh=False, grad_func=None):
        r"""
        Main function for mesh extraction from scalar field using FlexiCubes. This function converts
        discrete signed distance fields, encoded on voxel grids and additional per-cube parameters,
        to triangle or tetrahedral meshes using a differentiable operation as described in
        `Flexible Isosurface Extraction for Gradient-Based Mesh Optimization`_. FlexiCubes enhances
        mesh quality and geometric fidelity by adjusting the surface representation based on gradient
        optimization. The output surface is differentiable with respect to the input vertex positions,
        scalar field values, and weight parameters.
```

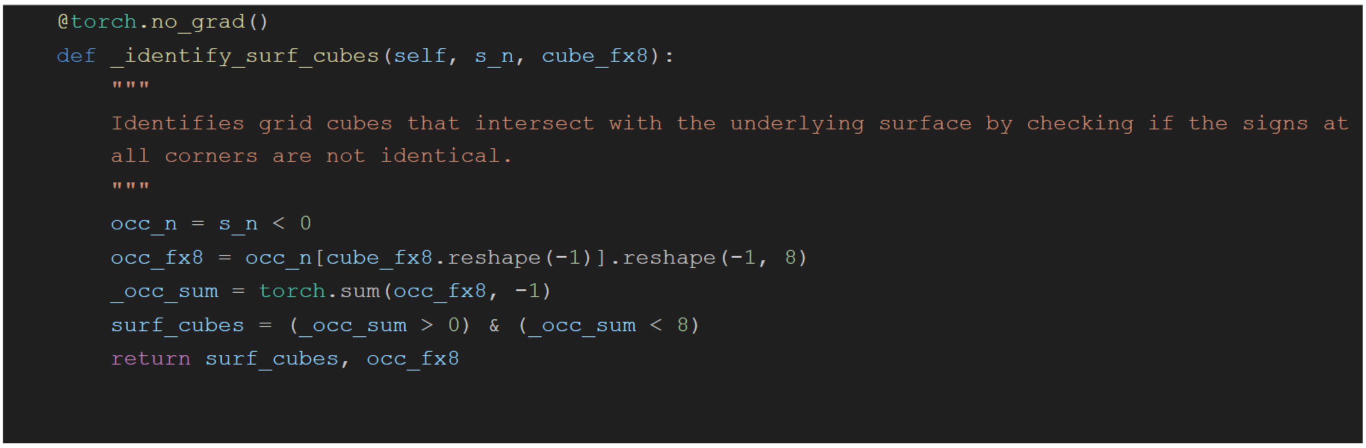
x_nx3 : 125,3

s_n : 125,1

cube_fx8 (torch.Tensor): Indices of 8 vertices for each cube in the voxel grid.

cube[0] -> tensor([0, 25, 5, 30, 1, 26, 6, 31], device='cuda:0')

1. surf_cubes, occ_fx8 = self._identify_surf_cubes(s_n, cube_fx8)



```
    @torch.no_grad()
    def _identify_surf_cubes(self, s_n, cube_fx8):
        """
        Identifies grid cubes that intersect with the underlying surface by checking if the signs at
        all corners are not identical.
        """
        occ_n = s_n < 0
        occ_fx8 = occ_n[cube_fx8.reshape(-1)].reshape(-1, 8)
        _occ_sum = torch.sum(occ_fx8, -1)
        surf_cubes = (_occ_sum > 0) & (_occ_sum < 8)
        return surf_cubes, occ_fx8
```

```
ipdb> surf_cubes.shape
torch.Size([64])
ipdb> surf_cubes
tensor([False, False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False, False,
         True,  True,  True,  True,  True,  True,  True,  True, False, False,
        False, False, False, False, False, False,  True,  True,  True,  True,
         True,  True,  True,  True, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False, False,
        False, False, False, False], device='cuda:0')
```