```
# Copyright (c) Facebook, Inc. and its affiliates. All rights reserved.
```

Fit a mesh via rendering

This tutorial shows how to:

- Load a mesh and textures from an `.obj` file.
- Create a synthetic dataset by rendering a textured mesh from multiple viewpoints
- Fit a mesh to the observed synthetic images using differential silhouette rendering
- Fit a mesh and its textures using differential textured rendering

0. Install and Import modules

Ensure `torch` and `torchvision` are installed. If `pytorch3d` is not installed, install it using the following cell:

```
import os
import sys
import torch
need_pytorch3d=False
try:
    import pytorch3d
except ModuleNotFoundError:
    need_pytorch3d=True
if need_pytorch3d:
    if torch.__version__.startswith("1.7") and sys.platform.startswith("linux"):
        # We try to install PyTorch3D via a released wheel.
        version_str="".join([
            f"py3{sys.version_info.minor}_cu",
            torch.version.cuda.replace(".",""),
            f"_pyt{torch.__version__[0:5:2]}"
        ])
        !pip install pytorch3d -f https://dl.fbaipublicfiles.com/pytorch3d/packagi
    else:
        # We try to install PyTorch3D from source.
        !curl -LO https://github.com/NVIDIA/cub/archive/1.10.0.tar.gz
        !tar xzf 1.10.0.tar.gz
        os.environ["CUB_HOME"] = os.getcwd() + "/cub-1.10.0"
        !pip install 'git+https://github.com/facebookresearch/pytorch3d.git@stable
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Curre
                                 Dload  Upload   Total   Spent    Left  Speed
100   118  100   118    0     0    737      0 --:--:-- --:--:-- --:--:--   73
100  404k    0  404k    0     0  1062k      0 --:--:-- --:--:-- --:--:-- 1062
Collecting git+https://github.com/facebookresearch/pytorch3d.git@stable
  Cloning https://github.com/facebookresearch/pytorch3d.git (to revision stab
  Running command git clone -q https://github.com/facebookresearch/pytorch3d.
  Running command git checkout -q 3c15a6c2469249c8b90a4f3e41e34350b8051b92
```

```
Collecting fvcore
  Downloading https://files.pythonhosted.org/packages/2d/4f/be471eb071ad04378
     |████████████████████████████████| 51kB 7.4MB/s
Collecting iopath
  Downloading https://files.pythonhosted.org/packages/21/d0/22104caed16fa4138
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-package
Collecting yacs>=0.1.6
  Downloading https://files.pythonhosted.org/packages/38/4f/fe9a4d472aa867878
Collecting pyyaml>=5.1
  Downloading https://files.pythonhosted.org/packages/7a/a5/393c087efdc78091a
     |████████████████████████████████| 645kB 35.6MB/s
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: termcolor>=1.1 in /usr/local/lib/python3.7/dis
Requirement already satisfied: Pillow in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: tabulate in /usr/local/lib/python3.7/dist-pack
Collecting portalocker
  Downloading https://files.pythonhosted.org/packages/68/33/cb524f4de29850992
Building wheels for collected packages: pytorch3d, fvcore
  Building wheel for pytorch3d (setup.py) ... done
  Created wheel for pytorch3d: filename=pytorch3d-0.4.0-cp37-cp37m-linux_x86_
  Stored in directory: /tmp/pip-ephem-wheel-cache-woiphfov/wheels/89/69/08/d8
  Building wheel for fvcore (setup.py) ... done
  Created wheel for fvcore: filename=fvcore-0.1.5.post20210508-cp37-none-any.
  Stored in directory: /root/.cache/pip/wheels/71/1c/b4/7e804154268b3de89058a
Successfully built pytorch3d fvcore
Installing collected packages: pyyaml, yacs, portalocker, iopath, fvcore, pyt
  Found existing installation: PyYAML 3.13
    Uninstalling PyYAML-3.13:
      Successfully uninstalled PyYAML-3.13
Successfully installed fvcore-0.1.5.post20210508 iopath-0.1.8 portalocker-2.3
```

```
import os
import torch
import matplotlib.pyplot as plt

from pytorch3d.utils import ico_sphere
import numpy as np
from tqdm.notebook import tqdm

# Util function for loading meshes
from pytorch3d.io import load_objs_as_meshes, save_obj

from pytorch3d.loss import (
    chamfer_distance,
    mesh_edge_loss,
    mesh_laplacian_smoothing,
    mesh_normal_consistency,
)

# Data structures and functions for rendering
from pytorch3d.structures import Meshes
from pytorch3d.renderer import (
    look_at_view_transform,
    OpenGLPerspectiveCameras,
    PointLights,
    DirectionalLights,
    Materials,
```

```
    RasterizationSettings,
    MeshRenderer,
    MeshRasterizer,
    SoftPhongShader,
    SoftSilhouetteShader,
    SoftPhongShader,
    TexturesVertex
)

# add path for demo utils functions
import sys
import os
sys.path.append(os.path.abspath(''))
```

If using **Google Colab**, fetch the utils file for plotting image grids:

```
!wget https://raw.githubusercontent.com/facebookresearch/pytorch3d/master/docs/tutc
from plot_image_grid import image_grid
```

```
--2021-05-08 12:27:34--  https://raw.githubusercontent.com/facebookresearch/p
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.10
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.1
HTTP request sent, awaiting response... 200 OK
Length: 1472 (1.4K) [text/plain]
Saving to: ‘plot_image_grid.py’

plot_image_grid.py  100%[===================>]   1.44K  --.-KB/s    in 0s

2021-05-08 12:27:34 (34.7 MB/s) - ‘plot_image_grid.py’ saved [1472/1472]
```

OR if running **locally** uncomment and run the following cell:

```
#  from utils.plot_image_grid import image_grid
```

1. Load a mesh and texture file

Load an `.obj` file and its associated `.mtl` file and create a **Textures** and **Meshes** object.

Meshes is a unique datastructure provided in PyTorch3D for working with batches of meshes of different sizes.

TexturesVertex is an auxiliary datastructure for storing vertex rgb texture information about meshes.

Meshes has several class methods which are used throughout the rendering pipeline.

If running this notebook using **Google Colab**, run the following cell to fetch the mesh obj and texture files and save it at the path `data/cow_mesh`: If running locally, the data is already

available at the correct path.

```
!mkdir -p data/cow_mesh
!wget -P data/cow_mesh https://dl.fbaipublicfiles.com/pytorch3d/data/cow_mesh/cow.o
!wget -P data/cow_mesh https://dl.fbaipublicfiles.com/pytorch3d/data/cow_mesh/cow.m
!wget -P data/cow_mesh https://dl.fbaipublicfiles.com/pytorch3d/data/cow_mesh/cow_t
```

```
--2021-05-08 12:27:40--  https://dl.fbaipublicfiles.com/pytorch3d/data/cow_me
Resolving dl.fbaipublicfiles.com (dl.fbaipublicfiles.com)... 104.22.75.142, 1
Connecting to dl.fbaipublicfiles.com (dl.fbaipublicfiles.com)|104.22.75.142|:
HTTP request sent, awaiting response... 200 OK
Length: 330659 (323K) [text/plain]
Saving to: 'data/cow_mesh/cow.obj'

cow.obj             100%[===================>] 322.91K  1.00MB/s    in 0.3s

2021-05-08 12:27:40 (1.00 MB/s) - 'data/cow_mesh/cow.obj' saved [330659/33065

--2021-05-08 12:27:40--  https://dl.fbaipublicfiles.com/pytorch3d/data/cow_me
Resolving dl.fbaipublicfiles.com (dl.fbaipublicfiles.com)... 104.22.75.142, 1
Connecting to dl.fbaipublicfiles.com (dl.fbaipublicfiles.com)|104.22.75.142|:
HTTP request sent, awaiting response... 200 OK
Length: 155 [text/plain]
Saving to: 'data/cow_mesh/cow.mtl'

cow.mtl             100%[===================>]     155  --.-KB/s    in 0s

2021-05-08 12:27:41 (31.6 MB/s) - 'data/cow_mesh/cow.mtl' saved [155/155]

--2021-05-08 12:27:41--  https://dl.fbaipublicfiles.com/pytorch3d/data/cow_me
Resolving dl.fbaipublicfiles.com (dl.fbaipublicfiles.com)... 104.22.75.142, 1
Connecting to dl.fbaipublicfiles.com (dl.fbaipublicfiles.com)|104.22.75.142|:
HTTP request sent, awaiting response... 200 OK
Length: 78699 (77K) [image/png]
Saving to: 'data/cow_mesh/cow_texture.png'

cow_texture.png     100%[===================>]  76.85K   469KB/s    in 0.2s

2021-05-08 12:27:42 (469 KB/s) - 'data/cow_mesh/cow_texture.png' saved [78699
```

```
# Setup
if torch.cuda.is_available():
    device = torch.device("cuda:0")
    torch.cuda.set_device(device)
else:
    device = torch.device("cpu")

# Set paths
DATA_DIR = "./data"
obj_filename = os.path.join(DATA_DIR, "cow_mesh/cow.obj")

# Load obj file
mesh = load_objs_as_meshes([obj_filename], device=device)

# We scale normalize and center the target mesh to fit in a sphere of radius 1
# centered at (0,0,0). (scale, center) will be used to bring the predicted mesh
```

```
# to its original center and scale.  Note that normalizing the target mesh,
# speeds up the optimization but is not necessary!
verts = mesh.verts_packed()
N = verts.shape[0]
center = verts.mean(0)
scale = max((verts - center).abs().max(0)[0])
mesh.offset_verts_(-center)
mesh.scale_verts_((1.0 / float(scale)));
```

2. Dataset Creation

We sample different camera positions that encode multiple viewpoints of the cow. We create a renderer with a shader that performs texture map interpolation. We render a synthetic dataset of images of the textured cow mesh from multiple viewpoints.

```
# the number of different viewpoints from which we want to render the mesh.
num_views = 20

# Get a batch of viewing angles.
elev = torch.linspace(0, 360, num_views)
azim = torch.linspace(-180, 180, num_views)

# Place a point light in front of the object. As mentioned above, the front of
# the cow is facing the -z direction.
lights = PointLights(device=device, location=[[0.0, 0.0, -3.0]])

# Initialize an OpenGL perspective camera that represents a batch of different
# viewing angles. All the cameras helper methods support mixed type inputs and
# broadcasting. So we can view the camera from the a distance of dist=2.7, and
# then specify elevation and azimuth angles for each viewpoint as tensors.
R, T = look_at_view_transform(dist=2.7, elev=elev, azim=azim)
cameras = OpenGLPerspectiveCameras(device=device, R=R, T=T)

# We arbitrarily choose one particular view that will be used to visualize
# results
camera = OpenGLPerspectiveCameras(device=device, R=R[None, 1, ...],
                                  T=T[None, 1, ...])

# Define the settings for rasterization and shading. Here we set the output
# image to be of size 128X128. As we are rendering images for visualization
# purposes only we will set faces_per_pixel=1 and blur_radius=0.0. Refer to
# rasterize_meshes.py for explanations of these parameters.  We also leave
# bin_size and max_faces_per_bin to their default values of None, which sets
# their values using heuristics and ensures that the faster coarse-to-fine
# rasterization method is used.  Refer to docs/notes/renderer.md for an
# explanation of the difference between naive and coarse-to-fine rasterization.
raster_settings = RasterizationSettings(
    image_size=128,
    blur_radius=0.0,
    faces_per_pixel=1,
)

# Create a Phong renderer by composing a rasterizer and a shader. The textured
```

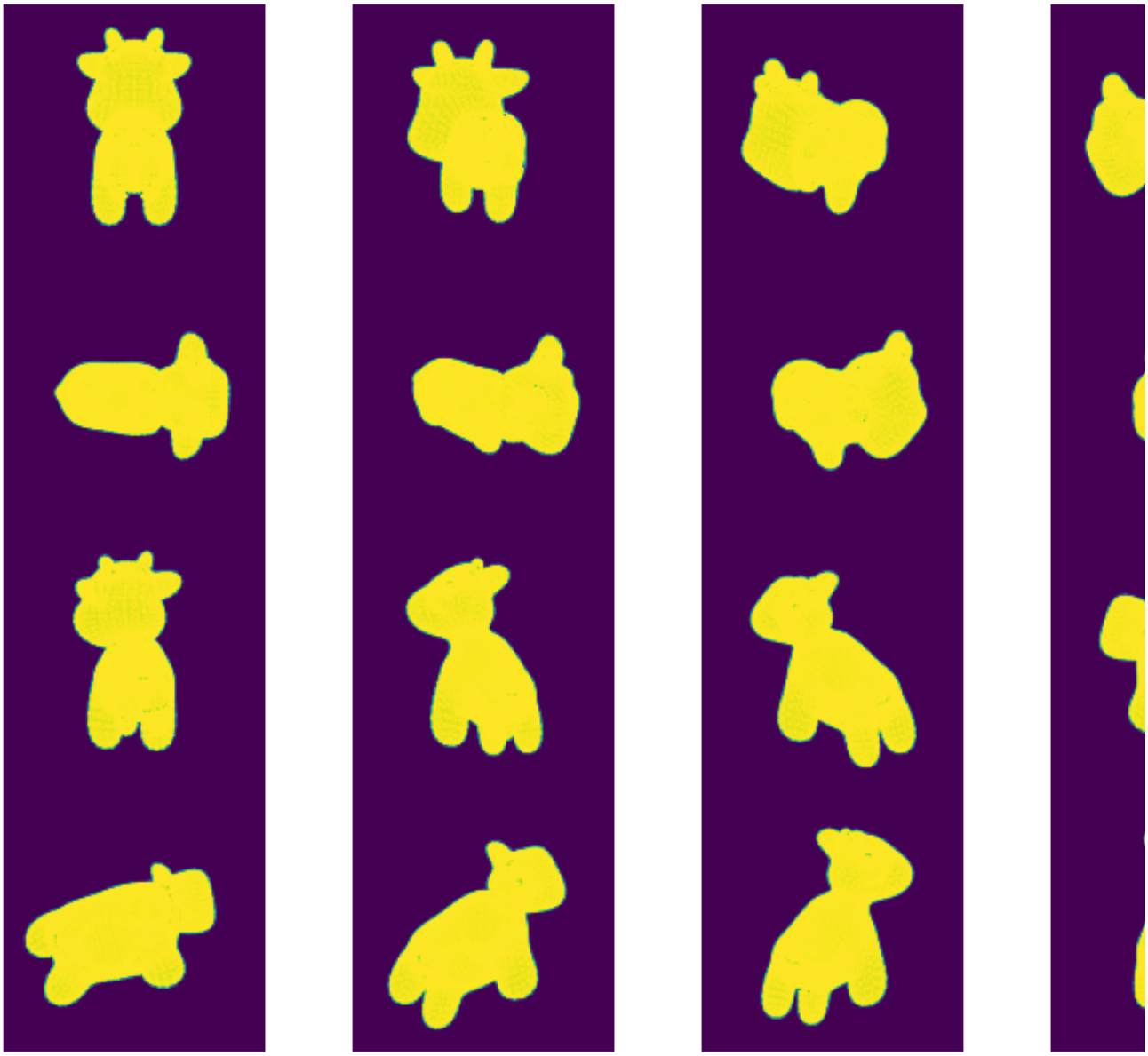
```
# Create a Phong renderer by composing a rasterizer and a shader. The textured
# Phong shader will interpolate the texture uv coordinates for each vertex,
# sample from a texture image and apply the Phong lighting model
renderer = MeshRenderer(
    rasterizer=MeshRasterizer(
        cameras=camera,
        raster_settings=raster_settings
    ),
    shader=SoftPhongShader(
        device=device,
        cameras=camera,
        lights=lights
    )
)

# Create a batch of meshes by repeating the cow mesh and associated textures.
# Meshes has a useful `extend` method which allows us do this very easily.
# This also extends the textures.
meshes = mesh.extend(num_views)

# Render the cow mesh from each viewing angle
target_images = renderer(meshes, cameras=cameras, lights=lights)

# Our multi-view cow dataset will be represented by these 2 lists of tensors,
# each of length num_views.
target_rgb = [target_images[i, ..., :3] for i in range(num_views)]
target_cameras = [OpenGLPerspectiveCameras(device=device, R=R[None, i, ...],
                                           T=T[None, i, ...]) for i in range(num_vi
```

Visualize the dataset:

```
# RGB images
image_grid(target_images.cpu().numpy(), rows=4, cols=5, rgb=True)
plt.show()
```

Later in this tutorial, we will fit a mesh to the rendered RGB images, as well as to just images of just the cow silhouette. For the latter case, we will render a dataset of silhouette images. Most shaders in PyTorch3D will output an alpha channel along with the RGB image as a 4th channel in an RGBA image. The alpha channel encodes the probability that each pixel belongs to the foreground of the object. We construct a soft silhouette shader to render this alpha channel.

```
# Rasterization settings for silhouette rendering
sigma = 1e-4
raster_settings_silhouette = RasterizationSettings(
    image_size=128,
    blur_radius=np.log(1. / 1e-4 - 1.)*sigma,
    faces_per_pixel=50,
)

# Silhouette renderer
renderer_silhouette = MeshRenderer(
    rasterizer=MeshRasterizer(
        cameras=camera,
        raster_settings=raster_settings_silhouette
    ),
    shader=SoftSilhouetteShader()
)

# Render silhouette images.  The 3rd channel of the rendering output is
# the alpha/silhouette channel
silhouette_images = renderer_silhouette(meshes, cameras=cameras, lights=lights)
target_silhouette = [silhouette_images[i, ..., 3] for i in range(num_views)]

# Visualize silhouette images
image_grid(silhouette_images.cpu().numpy(), rows=4, cols=5, rgb=False)
plt.show()
```



3. Mesh prediction via silhouette rendering

In the previous section, we created a dataset of images of multiple viewpoints of a cow. In this section, we predict a mesh by observing those target images without any knowledge of the ground truth cow mesh. We assume we know the position of the cameras and lighting.

We first define some helper functions to visualize the results of our mesh prediction:

```
# Show a visualization comparing the rendered predicted mesh to the ground truth
# mesh
def visualize_prediction(predicted_mesh, renderer=renderer_silhouette,
                         target_image=target_rgb[1], title='',
                         silhouette=False):
    inds = 3 if silhouette else range(3)
    with torch.no_grad():
        predicted_images = renderer(predicted_mesh)
    plt.figure(figsize=(20, 10))
    plt.subplot(1, 2, 1)
    plt.imshow(predicted_images[0, ..., inds].cpu().detach().numpy())
```

```
        plt.subplot(1, 2, 2)
        plt.imshow(target_image.cpu().detach().numpy())
        plt.title(title)
        plt.axis("off")

    # Plot losses as a function of optimization iteration
    def plot_losses(losses):
        fig = plt.figure(figsize=(13, 5))
        ax = fig.gca()
        for k, l in losses.items():
            ax.plot(l['values'], label=k + " loss")
        ax.legend(fontsize="16")
        ax.set_xlabel("Iteration", fontsize="16")
        ax.set_ylabel("Loss", fontsize="16")
        ax.set_title("Loss vs iterations", fontsize="16")
```

Starting from a sphere mesh, we will learn offsets of each vertex such that the predicted mesh silhouette is more similar to the target silhouette image at each optimization step. We begin by loading our initial sphere mesh:

```
# We initialize the source shape to be a sphere of radius 1.
src_mesh = ico_sphere(4, device)
```

We create a new differentiable renderer for rendering the silhouette of our predicted mesh:

```
# Rasterization settings for differentiable rendering, where the blur_radius
# initialization is based on Liu et al, 'Soft Rasterizer: A Differentiable
# Renderer for Image-based 3D Reasoning', ICCV 2019
sigma = 1e-4
raster_settings_soft = RasterizationSettings(
    image_size=128,
    blur_radius=np.log(1. / 1e-4 - 1.)*sigma,
    faces_per_pixel=50,
)

# Silhouette renderer
renderer_silhouette = MeshRenderer(
    rasterizer=MeshRasterizer(
        cameras=camera,
        raster_settings=raster_settings_soft
    ),
    shader=SoftSilhouetteShader()
)
```

We initialize settings, losses, and the optimizer that will be used to iteratively fit our mesh to the target silhouettes:

```
# Number of views to optimize over in each SGD iteration
num_views_per_iteration = 2
# Number of optimization steps
```

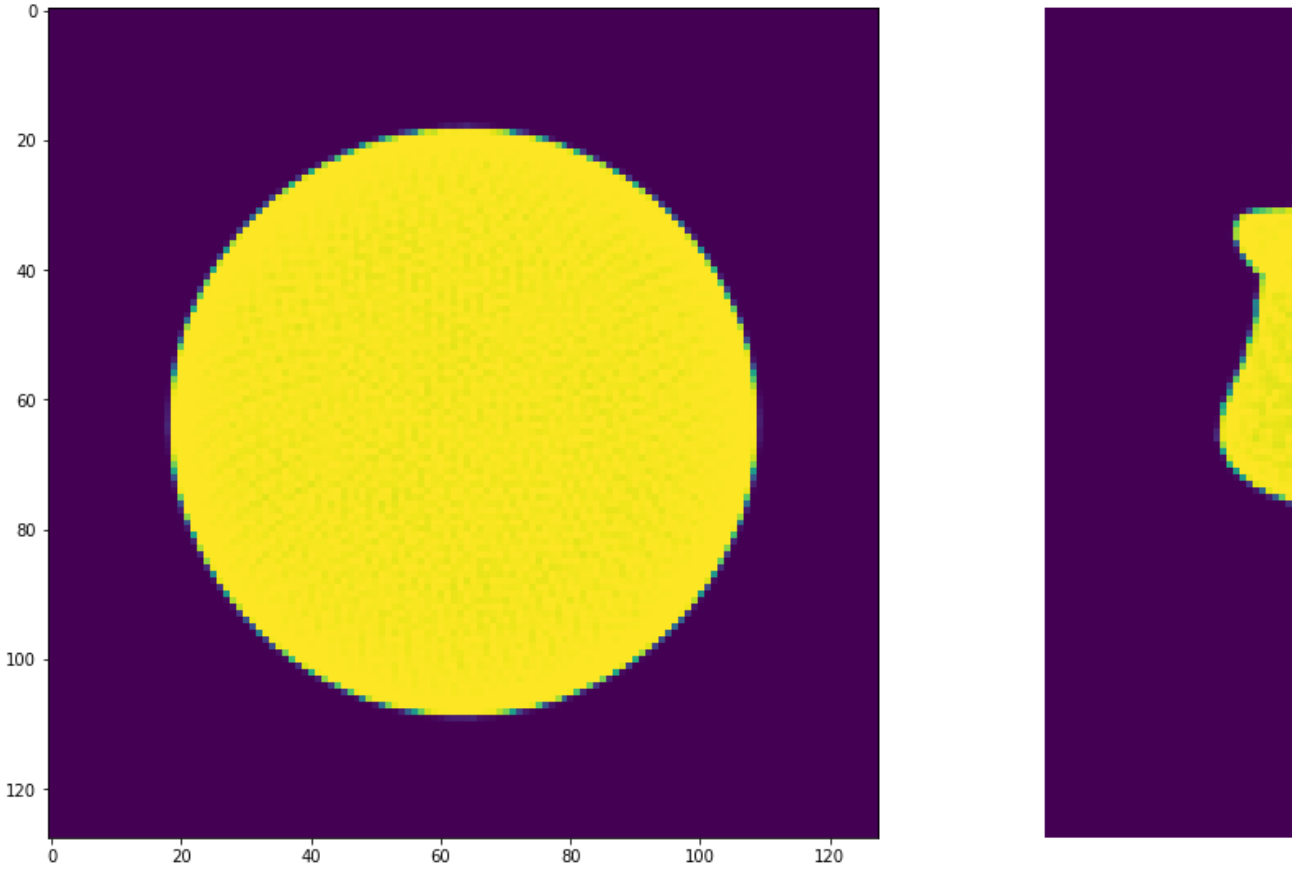
```
Niter = 2000
# Plot period for the losses
plot_period = 250

%matplotlib inline

# Optimize using rendered silhouette image loss, mesh edge loss, mesh normal
# consistency, and mesh laplacian smoothing
losses = {"silhouette": {"weight": 1.0, "values": []},
          "edge": {"weight": 1.0, "values": []},
          "normal": {"weight": 0.01, "values": []},
          "laplacian": {"weight": 1.0, "values": []},
         }

# Losses to smooth / regularize the mesh shape
def update_mesh_shape_prior_losses(mesh, loss):
    # and (b) the edge length of the predicted mesh
    loss["edge"] = mesh_edge_loss(mesh)

    # mesh normal consistency
    loss["normal"] = mesh_normal_consistency(mesh)

    # mesh laplacian smoothing
    loss["laplacian"] = mesh_laplacian_smoothing(mesh, method="uniform")

# We will learn to deform the source mesh by offsetting its vertices
# The shape of the deform parameters is equal to the total number of vertices in
# src_mesh
verts_shape = src_mesh.verts_packed().shape
deform_verts = torch.full(verts_shape, 0.0, device=device, requires_grad=True)

# The optimizer
optimizer = torch.optim.SGD([deform_verts], lr=1.0, momentum=0.9)
```

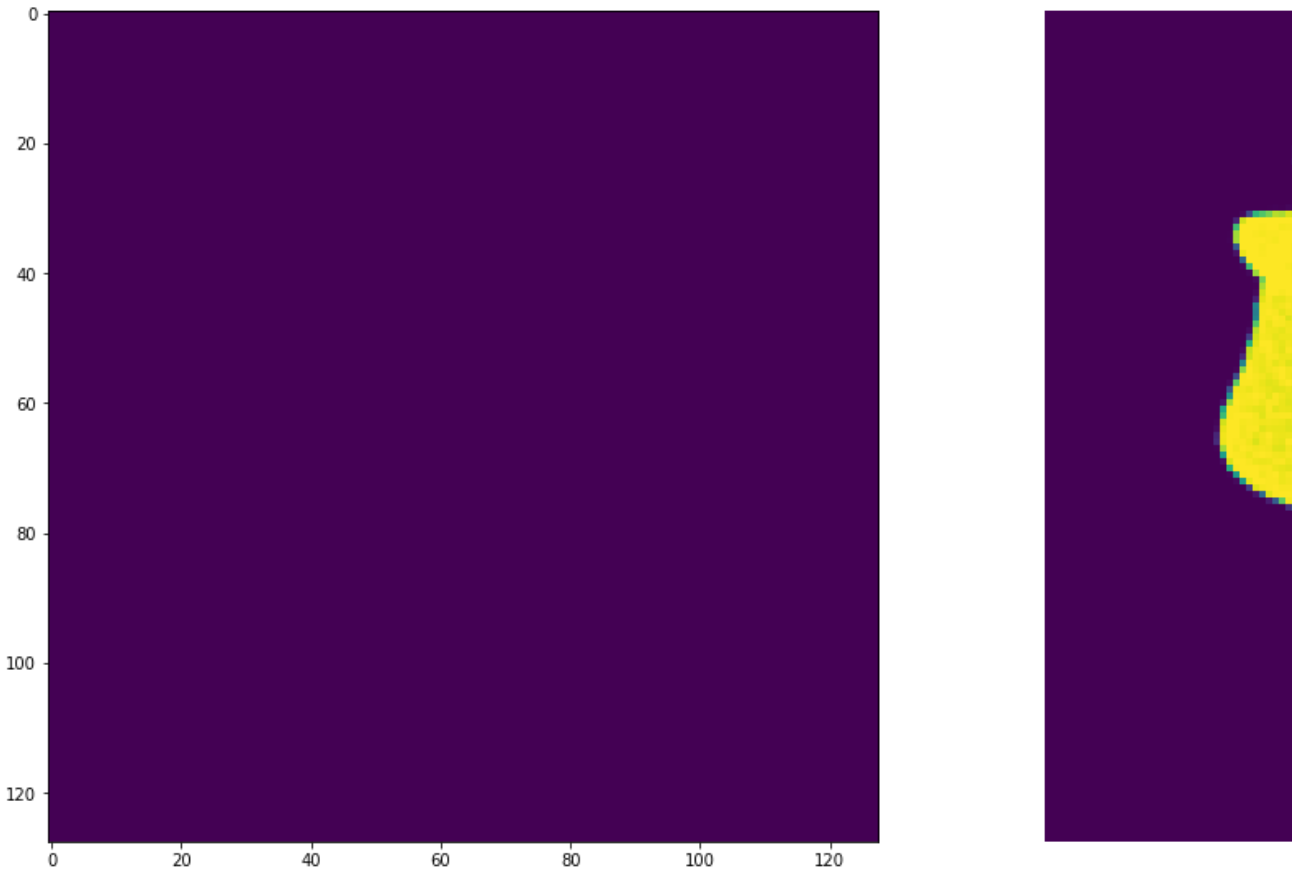
We write an optimization loop to iteratively refine our predicted mesh from the sphere mesh into a mesh that matches the silhouettes of the target images:

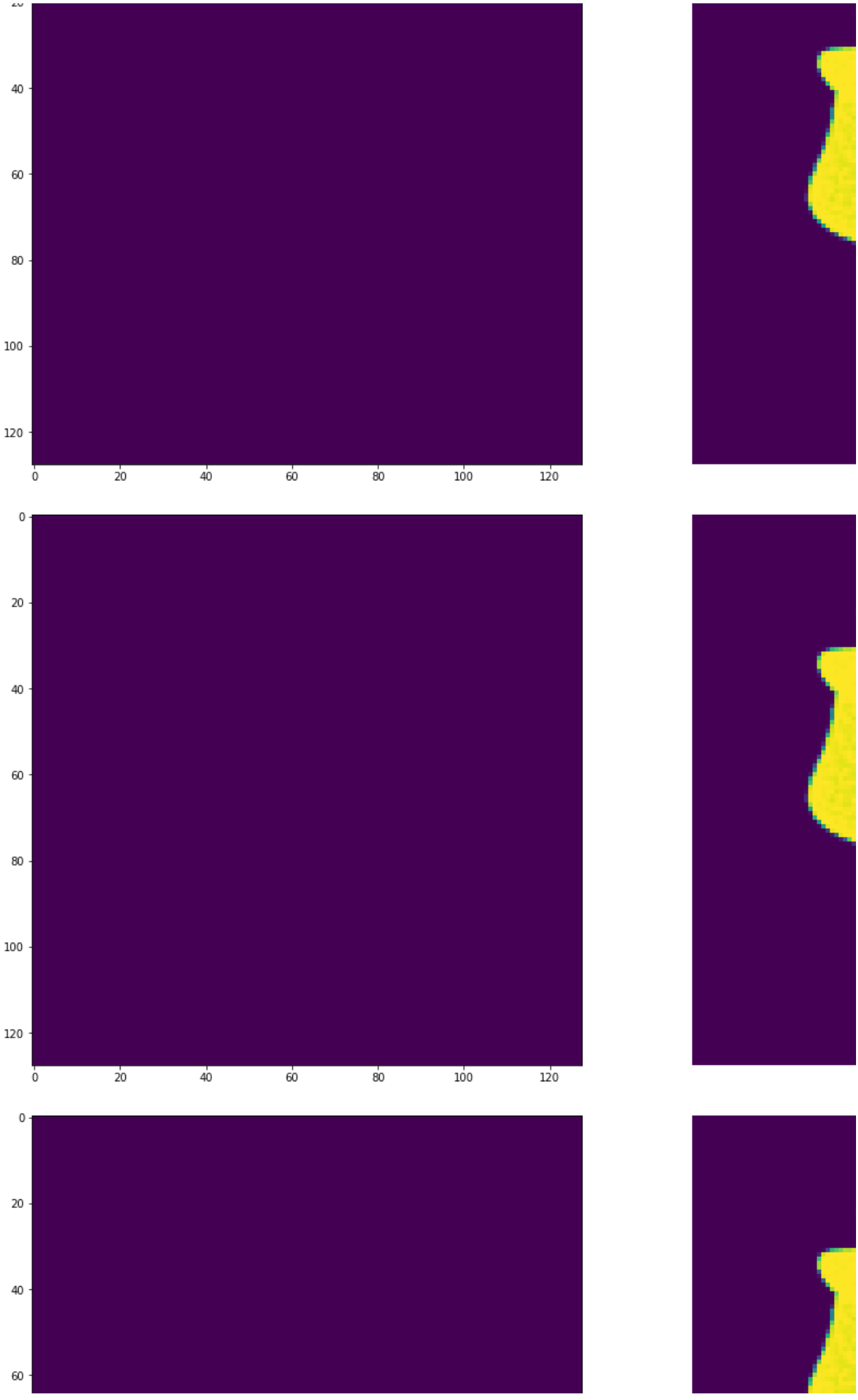
```
loop = tqdm(range(Niter))

for i in loop:
    # Initialize optimizer
    optimizer.zero_grad()

    # Deform the mesh
    new_src_mesh = src_mesh.offset_verts(deform_verts)

    # Losses to smooth /regularize the mesh shape
    loss = {k: torch.tensor(0.0, device=device) for k in losses}
    update_mesh_shape_prior_losses(new_src_mesh, loss)

    # Compute the average silhouette loss over two random views, as the average
    # squared L2 distance between the predicted silhouette and the target
    # silhouette from our dataset
```

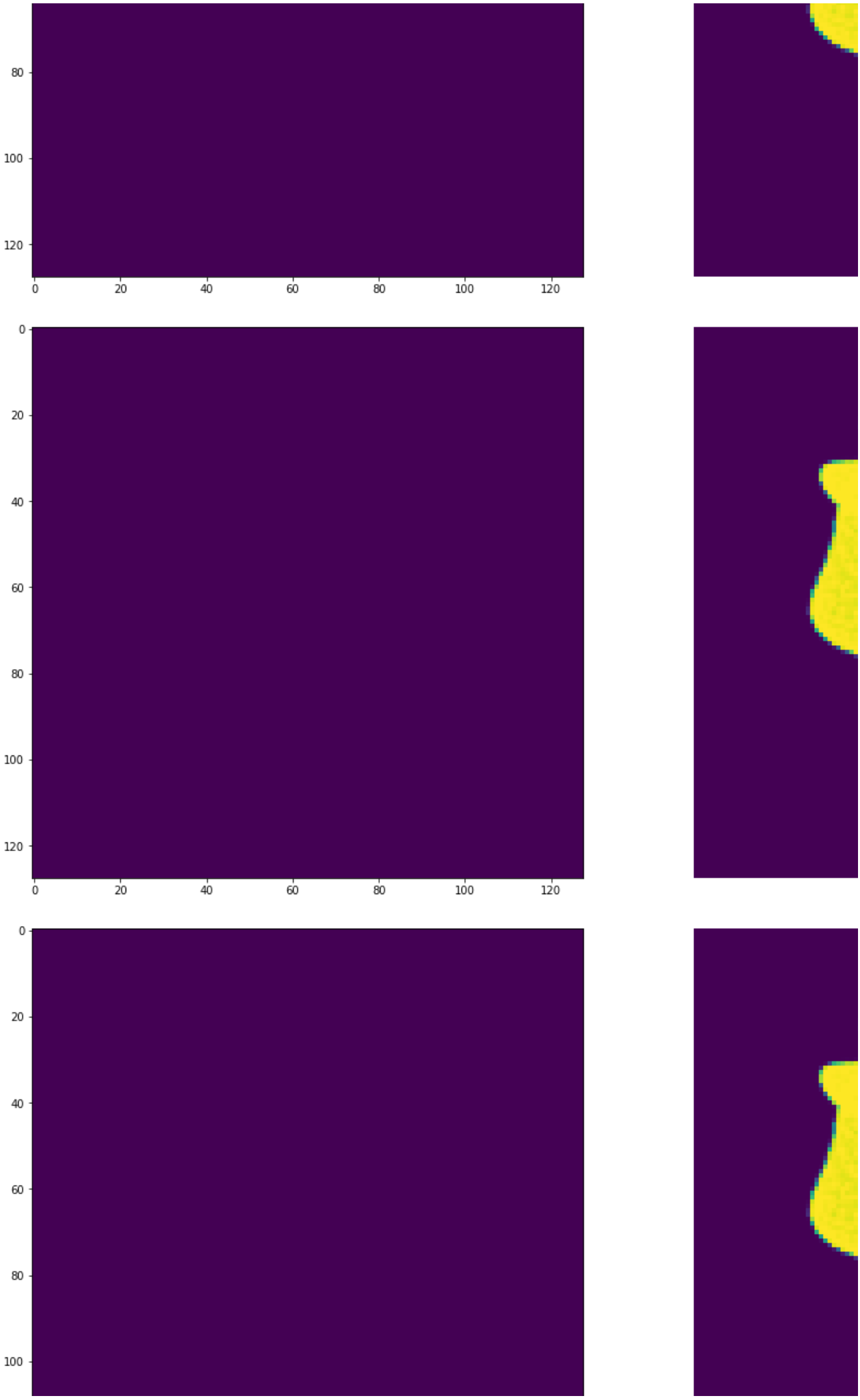
```
    for j in np.random.permutation(num_views).tolist()[:num_views_per_iteration]:
        images_predicted = renderer_silhouette(new_src_mesh, cameras=target_cameras
        predicted_silhouette = images_predicted[..., 3]
        loss_silhouette = ((predicted_silhouette - target_silhouette[j]) ** 2).mean
        loss["silhouette"] += loss_silhouette / num_views_per_iteration

    # Weighted sum of the losses
    sum_loss = torch.tensor(0.0, device=device)
    for k, l in loss.items():
        sum_loss += l * losses[k]["weight"]
        losses[k]["values"].append(l)

    # Print the losses
    loop.set_description("total_loss = %.6f" % sum_loss)

    # Plot mesh
    if i % plot_period == 0:
        visualize_prediction(new_src_mesh, title="iter: %d" % i, silhouette=True,
                             target_image=target_silhouette[1])

    # Optimization step
    sum_loss.backward()
    optimizer.step()
```

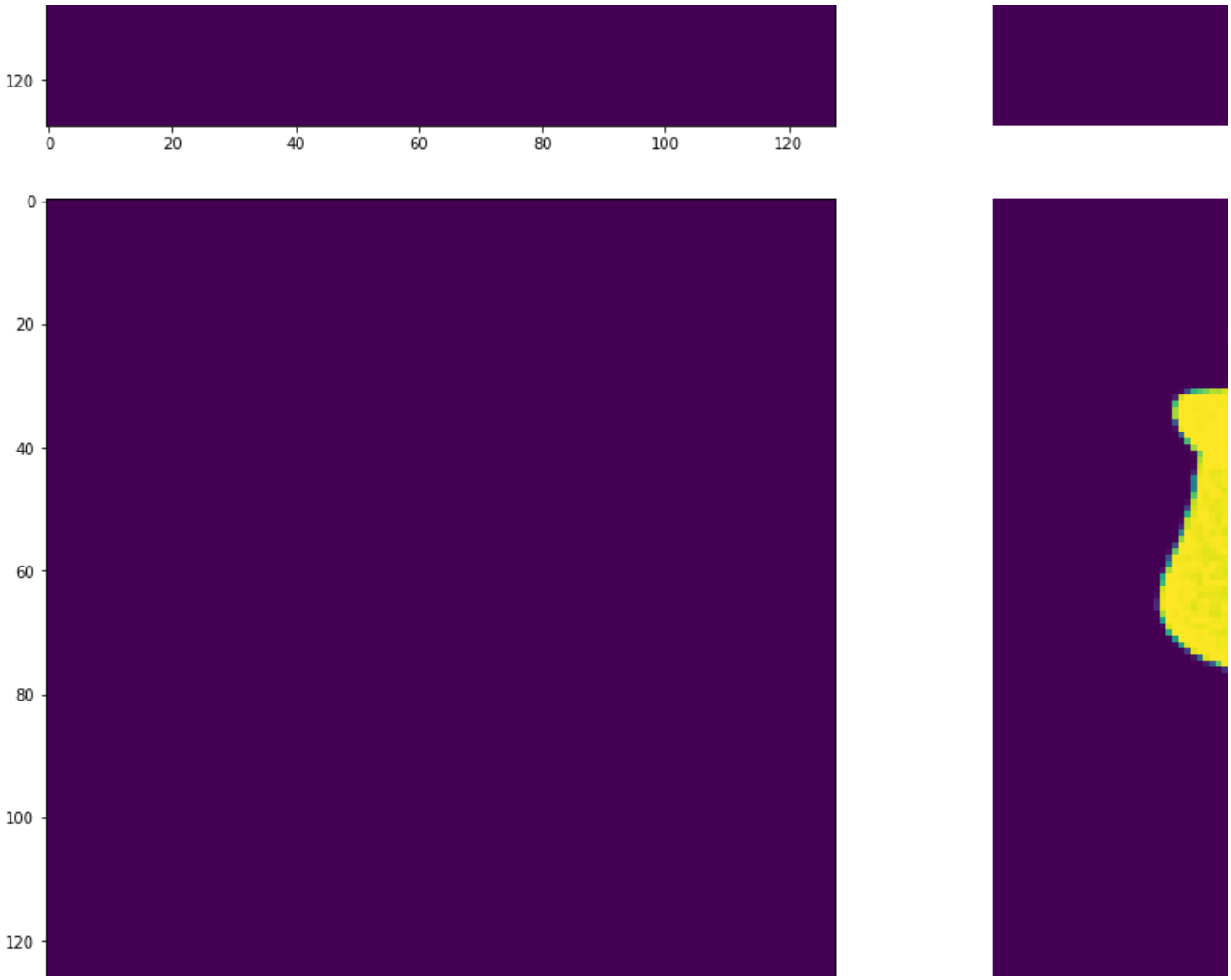
total_loss = nan: 100% 2000/2000 [01:56<00:00, 17.21it/s]






0
20
40
60
80
100
120


0
20
40
60
80
100
120




```
visualize_prediction(new_src_mesh, silhouette=True,
                     target_image=target_silhouette[1])
plot_losses(losses)
```

3. Mesh and texture prediction via textured rendering

We can predict both the mesh and its texture if we add an additional loss based on the comparing a predicted rendered RGB image to the target image. As before, we start with a sphere mesh. We learn both translational offsets and RGB texture colors for each vertex in the sphere mesh. Since our loss is based on rendered RGB pixel values instead of just the silhouette, we use a **SoftPhongShader** instead of a **SoftSilhouetteShader**.

```
# Rasterization settings for differentiable rendering, where the blur_radius
# initialization is based on Liu et al, 'Soft Rasterizer: A Differentiable
# Renderer for Image-based 3D Reasoning', ICCV 2019
sigma = 1e-4
raster_settings_soft = RasterizationSettings(
    image_size=128,
    blur_radius=np.log(1. / 1e-4 - 1.)*sigma,
    faces_per_pixel=50,
)

# Differentiable soft renderer using per vertex RGB colors for texture
renderer_textured = MeshRenderer(
    rasterizer=MeshRasterizer(
        cameras=camera,
        raster_settings=raster_settings_soft
    ),
    shader=SoftPhongShader(device=device,
        cameras=camera,
        lights=lights)
)
```

We initialize settings, losses, and the optimizer that will be used to iteratively fit our mesh to the target RGB images:

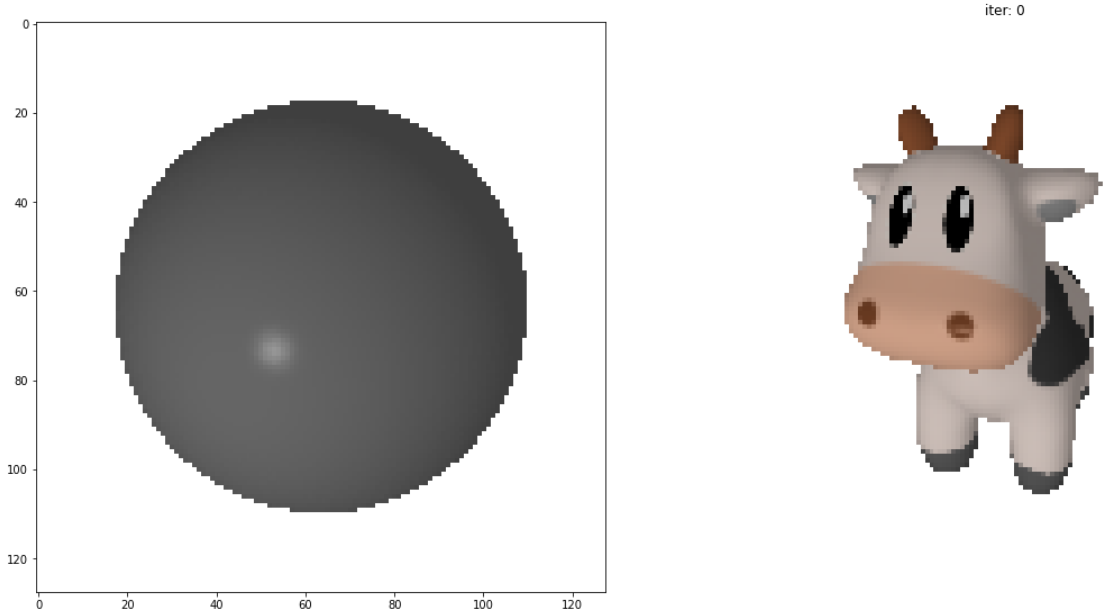
```
# Number of views to optimize over in each SGD iteration
num_views_per_iteration = 2
# Number of optimization steps
Niter = 2000
# Plot period for the losses
plot_period = 250

%matplotlib inline

# Optimize using rendered RGB image loss, rendered silhouette image loss, mesh
# edge loss, mesh normal consistency, and mesh laplacian smoothing
losses = {"rgb": {"weight": 1.0, "values": []},
          "silhouette": {"weight": 1.0, "values": []},
          "edge": {"weight": 1.0, "values": []},
          "normal": {"weight": 0.01, "values": []},
          "laplacian": {"weight": 1.0, "values": []},
         }

# We will learn to deform the source mesh by offsetting its vertices
# The shape of the deform parameters is equal to the total number of vertices in
# src_mesh
verts_shape = src_mesh.verts_packed().shape
deform_verts = torch.full(verts_shape, 0.0, device=device, requires_grad=True)

# We will also learn per vertex colors for our sphere mesh that define texture
# of the mesh
sphere_verts_rgb = torch.full([1, verts_shape[0], 3], 0.5, device=device, requires_

# The optimizer
optimizer = torch.optim.SGD([deform_verts, sphere_verts_rgb], lr=1.0, momentum=0.9)
```

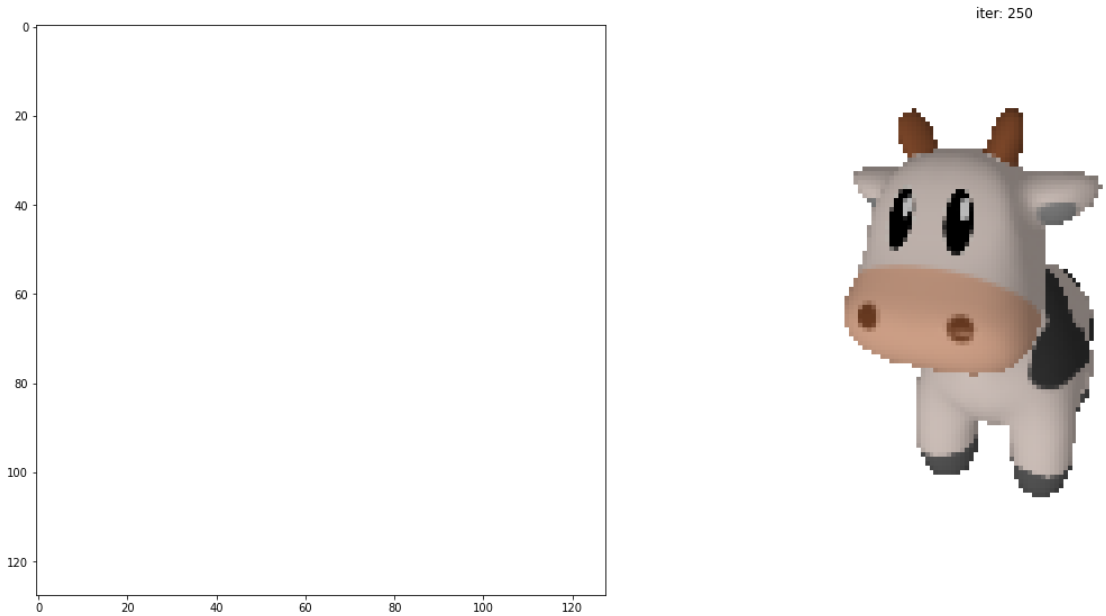
We write an optimization loop to iteratively refine our predicted mesh and its vertex colors from the sphere mesh into a mesh that matches the target images:

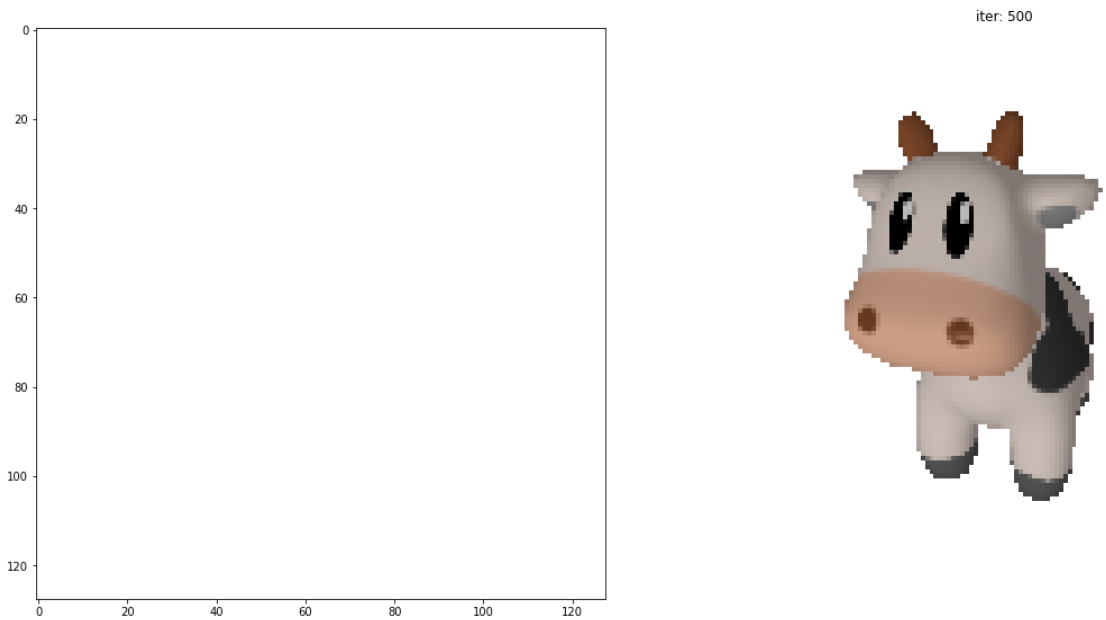
```
loop = tqdm(range(Niter))

for i in loop:
    # Initialize optimizer
    optimizer.zero_grad()

    # Deform the mesh
    new_src_mesh = src_mesh.offset_verts(deform_verts)

    # Add per vertex colors to texture the mesh
    new_src_mesh.textures = TexturesVertex(verts_features=sphere_verts_rgb)

    # Losses to smooth /regularize the mesh shape
    loss = {k: torch.tensor(0.0, device=device) for k in losses}
```

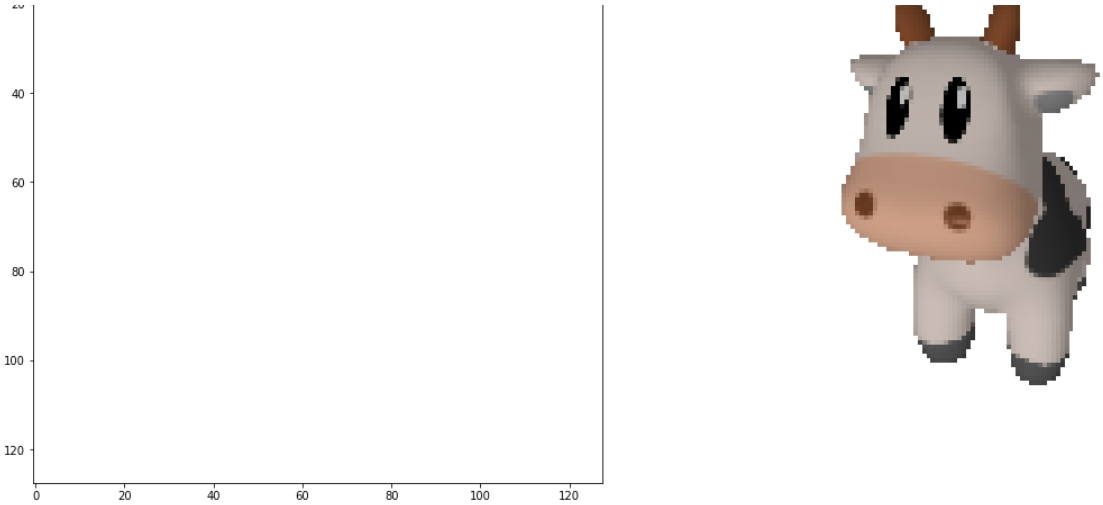
```
    update_mesh_shape_prior_losses(new_src_mesh, loss)

    # Randomly select two views to optimize over in this iteration.  Compared
    # to using just one view, this helps resolve ambiguities between updating
    # mesh shape vs. updating mesh texture
    for j in np.random.permutation(num_views).tolist()[:num_views_per_iteration]:
        images_predicted = renderer_textured(new_src_mesh, cameras=target_cameras[j

        # Squared L2 distance between the predicted silhouette and the target
        # silhouette from our dataset
        predicted_silhouette = images_predicted[..., 3]
        loss_silhouette = ((predicted_silhouette - target_silhouette[j]) ** 2).mean
        loss["silhouette"] += loss_silhouette / num_views_per_iteration

        # Squared L2 distance between the predicted RGB image and the target
        # image from our dataset
        predicted_rgb = images_predicted[..., :3]
        loss_rgb = ((predicted_rgb - target_rgb[j]) ** 2).mean()
        loss["rgb"] += loss_rgb / num_views_per_iteration

    # Weighted sum of the losses
    sum_loss = torch.tensor(0.0, device=device)
    for k, l in loss.items():
        sum_loss += l * losses[k]["weight"]
        losses[k]["values"].append(l)

    # Print the losses
    loop.set_description("total_loss = %.6f" % sum_loss)

    # Plot mesh
    if i % plot_period == 0:
        visualize_prediction(new_src_mesh, renderer=renderer_textured, title="iter:

    # Optimization step
    sum_loss.backward()
    optimizer.step()
```

total_loss = nan: 100% 2000/2000 [02:35<00:00, 12.86it/s]

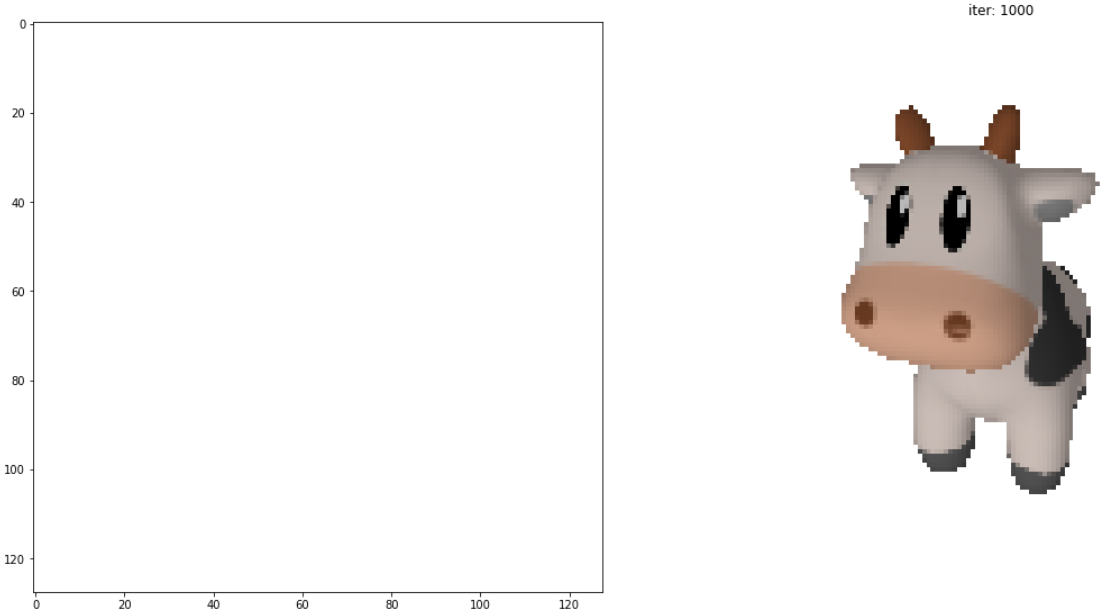




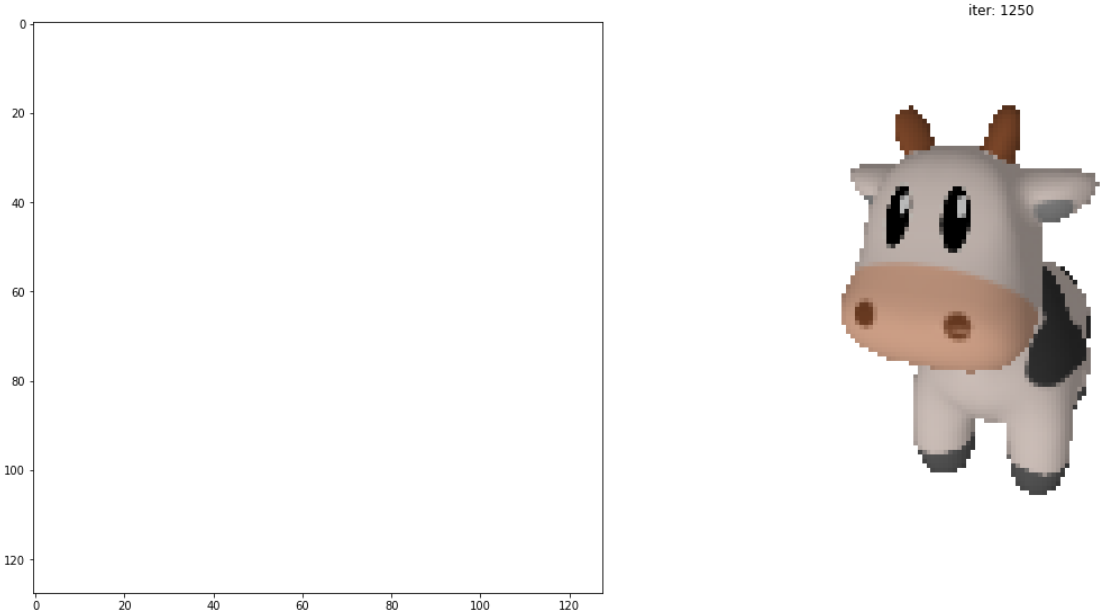




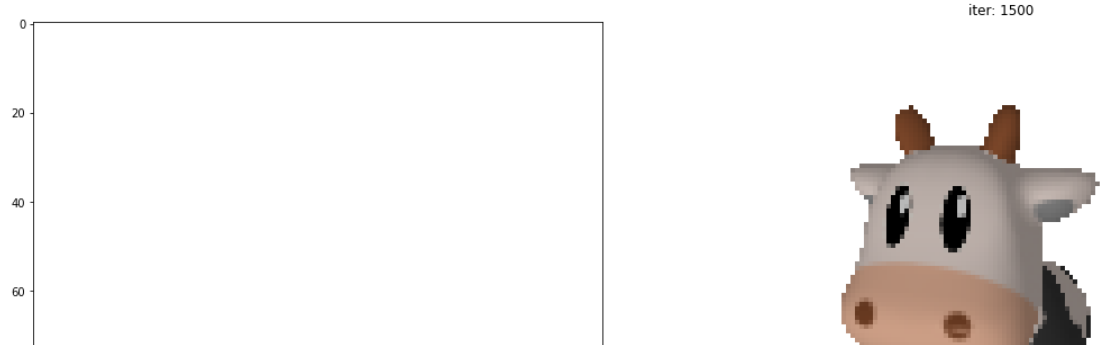
iter: 1000

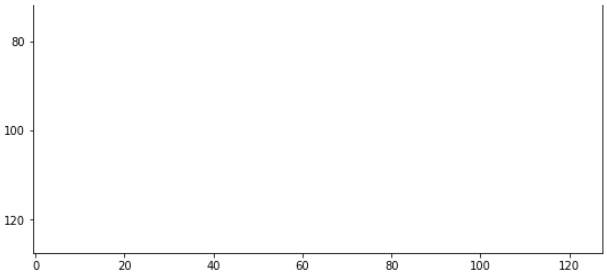


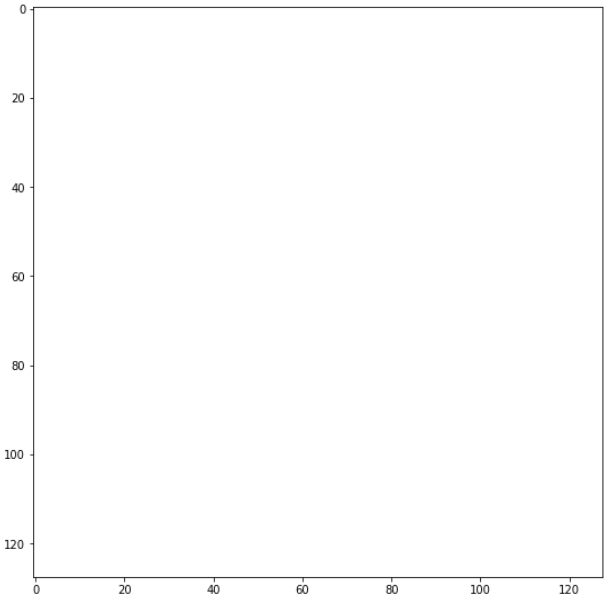
iter: 1250



iter: 1500

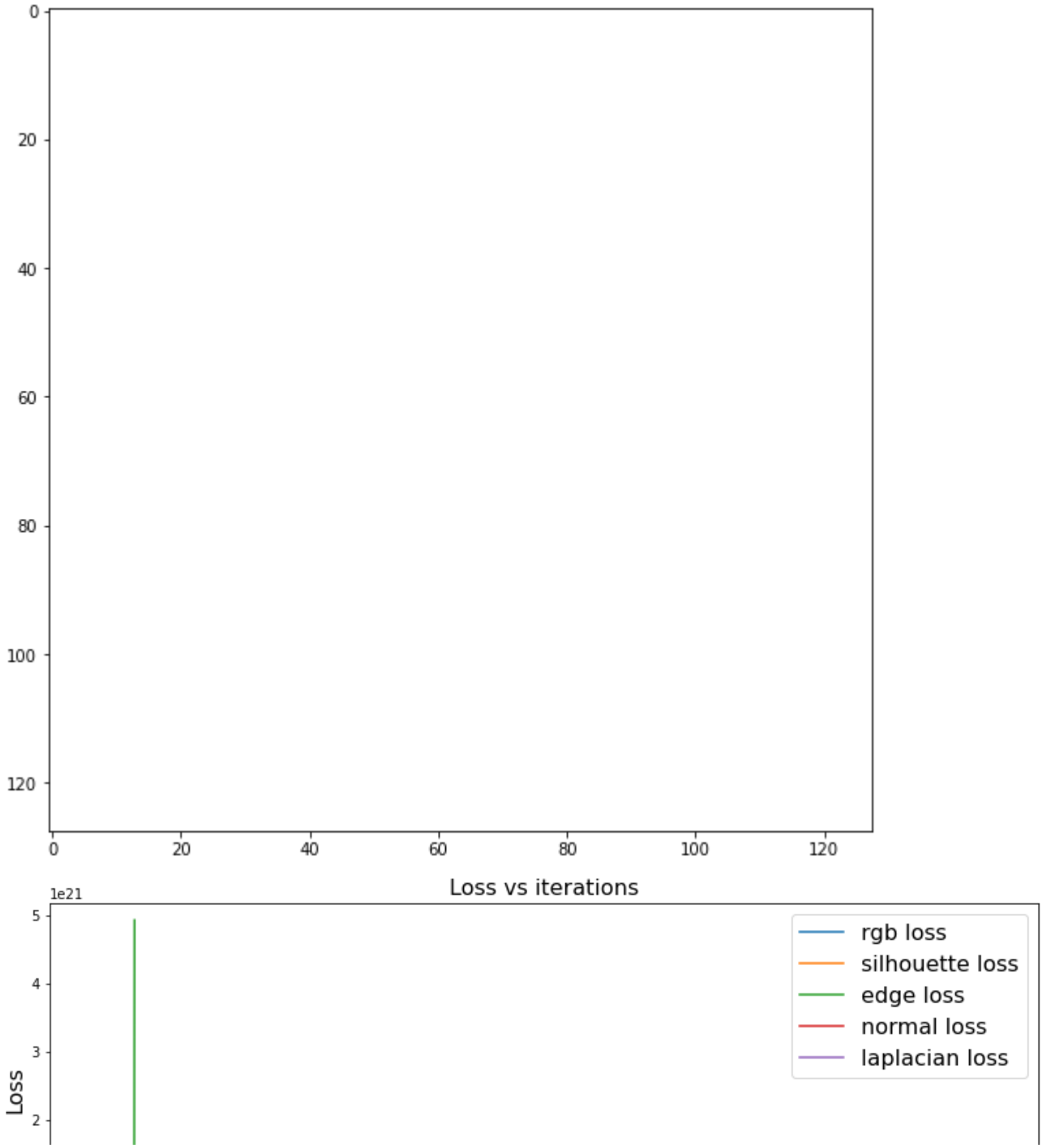



80
100
120
0
20
40
60
80
100
120


0
20
40
60
80
100
120
0
20
40
60
80
100
120

iter: 1750

```
visualize_prediction(new_src_mesh, renderer=renderer_textured, silhouette=False)
plot_losses(losses)
```



Save the final predicted mesh:

4. Save the final predicted mesh

```
# Fetch the verts and faces of the final predicted mesh
final_verts, final_faces = new_src_mesh.get_mesh_verts_faces(0)

# Scale normalize back to the original target size
final_verts = final_verts * scale + center

# Store the predicted mesh using save_obj
final_obj = os.path.join('./', 'final_model.obj')
save_obj(final_obj, final_verts, final_faces)
```

5. Conclusion

In this tutorial, we learned how to load a textured mesh from an obj file, create a synthetic dataset by rendering the mesh from multiple viewpoints. We showed how to set up an optimization loop to fit a mesh to the observed dataset images based on a rendered silhouette loss. We then augmented this optimization loop with an additional loss based on rendered RGB images, which allowed us to predict both a mesh and its texture.