Documentation

1. You need to run everything before the "Choose your algorithm" section. The exception is the "Connect Google Drive". It is for downloading sample images, you won't need them if you are going to use your own.
2. You can pick the algorithm you need and run it solely. Each of the blocks contains a line of form:

   ```
   source_image = PIL.Image.open("Cables.png").convert("RGB")
   ```

   This is where your image is loaded. You should upload your image into Google Colab, and change filename here. To upload an image just drag and drop it in the *files* section on the left.
3. After execution an image will appear in the notebook. If you right-click it a dropdown menu should appear. You can download the image from there.

Imports and Configurations

In [ ]:
```
import PIL.Image
import PIL.ImageOps
import PIL.ImageDraw as dw
import torch
import torchvision.models.vgg as vgg
from torch.nn.functional import interpolate, mse_loss, avg_pool2d, grid_sample, pad
from torch import nn
from torchvision.transforms.functional import to_tensor, to_pil_image
import math
import cmath
from ipywidgets import interact
import tqdm.notebook as tqdm
import itertools
import numpy as np
```

In [ ]:
```
device = "cuda"

style_layers = [1, 6, 12, 18]
```

Connect Google Drive

In [ ]:
```
# libraries for the files in google drive
from pydrive.auth import GoogleAuth
# from google.colab import drive
from pydrive.drive import GoogleDrive
from google.colab import auth
from oauth2client.client import GoogleCredentials
```

In [ ]:
```
auth.authenticate_user()
gauth = GoogleAuth()
gauth.credentials = GoogleCredentials.get_application_default()
drive = GoogleDrive(gauth)
files = {
    # "Kandinsky.jpg": '1Igk0jfnHukONLNYfLHnbjKzEoZspHf4-',
    "Cables.png": '1xhnMp8Cb2AMtmjbo6bKX1_SQqFGwG2Ah'
}

for filename, fileid in files.items():
    download = drive.CreateFile({'id': fileid})
    download.GetContentFile(filename)
```

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

In [ ]:
```
source_image = PIL.Image.open('/content/drive/My Drive/wood.webp').convert("RGB")
```

In [ ]:
```
source_image
```

Commons

In [ ]:
```
feature_extractor = vgg.vgg16(vgg.VGG16_Weights.IMAGENET1K_FEATURES).features.to(device)

for layer in feature_extractor:
    if hasattr(layer, "padding"):
        layer.padding = (0, 0)

def extract_features(input_tensor, mode="circular"):
    result = []
    for i, layer in enumerate(feature_extractor):
        if isinstance(layer, nn.Conv2d):
```

```
            input_tensor = pad(input_tensor, (1, 1, 1, 1), mode=mode)
        input_tensor = layer(input_tensor)
        if i in style_layers:
            result.append(input_tensor)
    return result
```

In [ ]:
```
def gram(x):
    n, c, h, w = x.shape
    return torch.einsum("nchw,nkhw->nck", x, x) / (h * w)
```

In [ ]:
```
def run_optimization(latent_tensor,
                     source_tensor,
                     uvmap,
                     number_of_iterations=80,
                     mode="circular"):
    optimizer = torch.optim.LBFGS([latent_tensor], history_size=5)
    with torch.no_grad():
        source_grams = [gram(t) for t in extract_features(source_tensor,
                                                          mode="reflect")]
    def closure():
        with torch.no_grad():
            latent_tensor.clamp_(0, 1)
            resolution = latent_tensor.shape[-1]
        generated_tensor = grid_sample(
            latent_tensor,
            uvmap[None],
            "nearest",
            "border",
            True,
        )
        optimizer.zero_grad()
        generated_grams = [gram(t) for t in extract_features(generated_tensor,
                                                             mode=mode)]
        loss_gram = sum(mse_loss(g, s) for g, s in zip(generated_grams,
                                                       source_grams))
        loss_tv = torch.abs(latent_tensor[:, :, 1:-1, 1:-1] -
                            avg_pool2d(latent_tensor,
                                       (3, 3),
                                       (1, 1))
                            ).mean()
        loss = loss_gram + loss_tv
        loss_gram.backward()
        return loss

    progress_bar = tqdm.trange(number_of_iterations)
    for stage in progress_bar:
        loss = optimizer.step(closure)
        progress_bar.set_description(f"loss = {loss.item():.3}")
```

Different Tiling Methods

In [ ]:
```
def create_uvmap_identity(resolution):
    x, y = torch.meshgrid(torch.linspace(-1, 1, resolution),
                          torch.linspace(-1, 1, resolution))

    xy = torch.stack([y, x], dim=0)
    return torch.movedim(xy, 0, -1).to(device)
```

In [ ]:
```
def create_uvmap_square_tiling(resolution, canvas_multiplier=2):
    x, y = torch.meshgrid(torch.linspace(-canvas_multiplier,
                                         canvas_multiplier,
                                         canvas_multiplier * resolution),
                          torch.linspace(-canvas_multiplier,
                                         canvas_multiplier,
                                         canvas_multiplier * resolution))

    xy = torch.stack([y, x], dim=0)
    xy = (xy + 1.0 + 2 * canvas_multiplier) % 2.0 - 1.0
    return torch.movedim(xy, 0, -1).to(device)
```

In [ ]:
```
def create_uvmap_edge_tiling(resolution, canvas_multiplier=2):
    x, y = torch.meshgrid(torch.linspace(-canvas_multiplier,
                                         canvas_multiplier,
                                         int(canvas_multiplier * resolution)),
                          torch.linspace(-canvas_multiplier,
                                         canvas_multiplier,
                                         int(canvas_multiplier * resolution)))

    xy = torch.stack([y, x], dim=0)
    xy = torch.clamp(xy, -1.0, 1.0)
    return torch.movedim(xy, 0, -1).to(device)
```

In [ ]:
```
to_pil_image(torch.movedim(create_uvmap_hexagonal_tiling(512), -1, 0)[[0, 0, 1]] * 0.5 + 0.5)
```

In [ ]:
```
def create_uvmap_spiral(resolution,
```

```
                        phase=0.0,
                        scale=2.0,
                        rotation=1,
                        focus=(0.0, 0.5),
                        aspect_ratio=1.0
                        ):
    x, y = torch.meshgrid(torch.linspace(-aspect_ratio,
                                         aspect_ratio,
                                         int(aspect_ratio*
                                             resolution)),
                          torch.linspace(-1.0, 1.0, resolution),
                          indexing="xy")

    complex_focus = focus[0] + 1j * focus[1]
    complex_transform = 1.0 / scale * cmath.exp(1j * rotation)
    complex_offset = complex_focus * (1.0 - complex_transform)

    x, y = x.cpu().data.numpy(), y.cpu().data.numpy()
    z = x + 1j * y
    uvmap = z.copy()

    for i in range(-4, 10):
        i = i + phase
        center = ((1.0 - complex_transform ** i) /
                  (1.0 - complex_transform) *
                  complex_offset)
        transform = complex_transform ** i
        radius = abs(scale) ** (-i)
        uvmap = np.where(abs(z - center) < radius,
                         (z - center) / transform,
                         uvmap)
    output = torch.stack([torch.tensor(uvmap.real),
                          torch.tensor(uvmap.imag)],
                         dim=-1)
    return output.to(device)
```

In [ ]:
```
@interact
def spiral(phase = 0.0):
    uvmap_spiral = torch.movedim(create_uvmap_spiral(256, phase=phase), -1, 0)
    display(to_pil_image(uvmap_spiral[[0, 0, 1]] * 0.5 + 0.5))
```

Generation Function

In [ ]:
```
def generate(source_image,
             uvmap_generator,
             resolution,
             uvmap_generator_latent=None,
             mode="circular"):
    source_image = to_tensor(source_image)
    source_image = source_image[None].to(device)

    if uvmap_generator_latent is None:
        uvmap_generator_latent = uvmap_generator

    result = torch.zeros((1, 3, resolution, resolution)).to(device)

    with torch.no_grad():
        n_stages = int(math.log2(resolution))
        for stage in range(n_stages):
            stage_resolution = resolution // (2 ** stage)
            stage_amplitude = 0.25 * (2 ** (n_stages - stage - 1))
            noise_low_resolution = (0.5 + (torch.rand((1, 3,
                                                       stage_resolution,
                                                       stage_resolution)) - 0.5)
                                    * stage_amplitude)
            noise = interpolate(noise_low_resolution, (resolution, resolution))
            result.data += noise.to(device)

    for i in range(4, 0, -1):
        with torch.no_grad():
            source_image_resized = interpolate(
                    source_image,
                    (resolution // i, resolution // i),
                    mode="area"
                )
            result = interpolate(
                    result.detach(),
                    (resolution // i, resolution // i)
                )
            result = result.detach().requires_grad_(True)


        uvmap = uvmap_generator(resolution // i)
        run_optimization(result, source_image_resized, uvmap, mode=mode)

        with torch.no_grad():
```

```
            uvmap_latent = uvmap_generator_latent(resolution // i)
            result.data = grid_sample(
                result,
                uvmap_latent[None],
                "nearest",
                "border",
                True)
        display_result = result # grid_sample(result, uvmap[None], "nearest", "border", True)
        display_result = torch.clamp(display_result, 0.0, 1.0)
        display(to_pil_image(display_result[0]))
    return to_pil_image(result[0])
```

Choose your algorithm

No Tiling

In [ ]:
```
# source_image = PIL.Image.open("ArrowsTheorem.png").convert("RGB")
uvmap_generator = lambda x: create_uvmap_identity(x)

output = generate(source_image, uvmap_generator, resolution=256, mode="reflect")
output
```

In [ ]:
```
uvmap = create_uvmap_identity(256)

display_result = grid_sample(to_tensor(output)[None].to(device),
                             uvmap[None],
                             "nearest",
                             "border",
                             True)
display_result = torch.clamp(display_result, 0.0, 1.0)
display(to_pil_image(display_result[0]))
```

Square Tiling

In [ ]:
```
uvmap_generator = lambda x: create_uvmap_identity(x)

output = generate(source_image, uvmap_generator, resolution=256)
output
```

In [ ]:
```
uvmap = create_uvmap_square_tiling(1024, canvas_multiplier=2)

display_result = grid_sample(to_tensor(output)[None].to(device),
                             uvmap[None],
                             "nearest",
                             "border",
                             True)
display_result = torch.clamp(display_result, 0.0, 1.0)
display(to_pil_image(display_result[0]))
```

In [ ]:
```
resolution = 256

canvas_multiplier = 2

uvmap = create_uvmap_square_tiling(resolution,
                                   canvas_multiplier=canvas_multiplier)

display_result = grid_sample(to_tensor(output)[None].to(device),
                             uvmap[None],
                             "nearest",
                             "border",
                             True)

display_result = torch.clamp(display_result, 0.0, 1.0)

uvmap_blurred = torch.moveaxis(
    avg_pool2d(torch.moveaxis(uvmap, -1, 0)[None], 5, 1, 2)[0],
    0, -1)

grid_outlines = (torch.max((uvmap_blurred - uvmap).abs(), dim=-1)[0] > 0.25)

y, x = torch.meshgrid(
    torch.linspace(-canvas_multiplier, canvas_multiplier, display_result.shape[2]),
    torch.linspace(-canvas_multiplier, canvas_multiplier, display_result.shape[3]),
)

grid_outlines = torch.min(grid_outlines, (x * x + y * y).to(device) < 2).float()

display_result = ((1.0 - display_result) * grid_outlines * 0.7 +
                  (1.0 - grid_outlines) * display_result)

pil_canvas = to_pil_image(display_result[0])
```

```
display(pil_canvas)
```

Hexagonal Tiling

In [ ]:
```
uvmap_generator_latent = lambda x: create_uvmap_hexagonal_tiling(
    x, crop_rectangle=(-1, -1, 1, 1)
)
uvmap_generator = lambda x: create_uvmap_hexagonal_tiling(x)

output = generate(source_image,
                  uvmap_generator,
                  uvmap_generator_latent=uvmap_generator_latent,
                  resolution=300)
output
```

In [ ]:
```
uvmap = create_uvmap_hexagonal_tiling(256,
                                      (-3, -3, 3, 3))

display_result = grid_sample(to_tensor(output)[None].to(device),
                             uvmap[None],
                             "nearest",
                             "border",
                             True)
display_result = torch.clamp(display_result, 0.0, 1.0)
display(to_pil_image(display_result[0]))
```

In [ ]:
```
resolution = 256

x_min = -2
y_min = -2
x_max = 2
y_max = 2

uvmap = create_uvmap_hexagonal_tiling(resolution,
                                      (x_min, y_min, x_max, y_max))

display_result = grid_sample(to_tensor(output)[None].to(device),
                             uvmap[None],
                             "nearest",
                             "border",
                             True)

display_result = torch.clamp(display_result, 0.0, 1.0)

uvmap_blurred = torch.moveaxis(
    avg_pool2d(torch.moveaxis(uvmap, -1, 0)[None], 5, 1, 2)[0],
    0, -1)

grid_outlines = (torch.max((uvmap_blurred - uvmap).abs(), dim=-1)[0] > 0.25)

y, x = torch.meshgrid(
    torch.linspace(y_min, y_max, display_result.shape[2]),
    torch.linspace(x_min, x_max, display_result.shape[3]),
)

grid_outlines = torch.min(grid_outlines, (x * x + y * y).to(device) < 1).float()

display_result = ((1.0 - display_result) * grid_outlines * 0.7 +
                  (1.0 - grid_outlines) * display_result)

pil_canvas = to_pil_image(display_result[0])
display(pil_canvas)
```

Zoom in loop

Generation

In [ ]:
```
uvmap_generator = lambda x: create_uvmap_spiral(x)

output = generate(source_image, uvmap_generator, resolution=500)
output
```

Demonstration

In [ ]:
```
uvmap = uvmap_generator(256)

display_result = grid_sample(to_tensor(output)[None].to(device),
                             uvmap[None],
                             "nearest",
                             "border",
                             True)
```

```
display_result = torch.clamp(display_result, 0.0, 1.0)
display(to_pil_image(display_result[0]))
```

Export

In [ ]:
```
frames = []

for phase in tqdm.tqdm(np.linspace(1.0, 0.0, 120)):
    uvmap = create_uvmap_spiral(500, phase=phase, aspect_ratio=2.0)

    display_result = grid_sample(to_tensor(output)[None].to(device),
                                 uvmap[None],
                                 "nearest",
                                 "border",
                                 True)
    display_result = avg_pool2d(display_result, (2, 2), (2, 2))
    display_result = torch.clamp(display_result, 0.0, 1.0)
    frames.append(to_pil_image(display_result[0]))


first_frame = frames[0]
first_frame.save("spiral.gif",
                 save_all=True,
                 append_images=frames[1:],
                 duration=25,
                 loop=0)
```

In [ ]:
```
frames_raw = []
for phase in tqdm.tqdm(np.linspace(1.0, 0.0, 60)):
    uvmap = create_uvmap_spiral(500, phase=phase, aspect_ratio=2.0)

    display_result = grid_sample(to_tensor(output)[None].to(device),
                                 uvmap[None],
                                 "nearest",
                                 "border",
                                 True)
    display_result = avg_pool2d(display_result, (2, 2), (2, 2))
    display_result = torch.clamp(display_result, 0.0, 1.0)
    frames_raw.append(display_result[0])


frames = [to_pil_image(a * 0.2 + b * 0.8) for a, b in
          zip(tqdm.tqdm(frames_raw[::2]), frames_raw[1::2])]

first_frame = frames[0]
first_frame.save("spiral.gif",
                 save_all=True,
                 append_images=frames[1:],
                 duration=40,
                 loop=0)
```

In [ ]:
```
first_frame
```

In [ ]:
```
!pip install nbconvert
```

In [ ]:
```
# !jupyter nbconvert --execute --to html "/content/drive/My Drive/results.ipynb"
!sudo apt-get install pandoc
```

In [ ]:
```
!sudo apt-get install texlive-xetex texlive-fonts-recommended texlive-plain-generic
```

In [ ]:
```
!jupyter nbconvert --execute --to html results.ipynb
```