Contents

1 Basic Test Results 2

2 README 4

3 answer q1.txt 5

4 answer q2.txt 6

5 answer q3.txt 7

6 sol3.py 8

7 imgs/aq.jpg 12

8 imgs/c.jpg 13

9 imgs/d.jpg 14

10 imgs/dude mask.jpg 15

11 imgs/e.jpg 16

12 imgs/mask aq.jpg 17

1 Basic Test Results

```
Archive:  /tmp/bodek.pPLy7U/impr/ex3_late/mottig/presubmission/submission
  inflating: current/README
  inflating: current/answer_q1.txt
  inflating: current/answer_q2.txt
  inflating: current/answer_q3.txt
   creating: current/imgs/
  inflating: current/imgs/aq.jpg
  inflating: current/imgs/c.jpg
  inflating: current/imgs/d.jpg
  inflating: current/imgs/dude_mask.jpg
  inflating: current/imgs/e.jpg
  inflating: current/imgs/mask_aq.jpg
  inflating: current/sol3.py
ex3 presubmission script

    Disclaimer
    ----------
    The purpose of this script is to make sure that your code is compliant
    with the exercise API and some of the requirements
    The script does not test the quality of your results.
    Don't assume that passing this script will guarantee that you will get
    a high grade in the exercise

=== Check Submission ===

login:  mottig

submitted files:
sol3.py
answer_q1.txt
answer_q2.txt
answer_q3.txt

=== Answers to questions ===

Answer to Q1:
As we learn in class, laplacian pyramids can use as a band-pass-filter.
By multiplying each level with some coefficient, we actually applying the BPF.
For example, multiplying the smallest level by 0 will filter the low frequencies,
in another words - LPF. multiplying the biggest one by zero will give us HPF. and any
other level and coefficients combining a BPF.

Answer to Q2:
Here is another frequencies filtering. For too small window we let all the details remain in
both images, which cause us to see all the edges as we let the high frequencies stay, so the
blending is of course not good enough.
For too big window, on the other hand, we creating a strict low pass filter, which will also
cause us to get only very blur details, like the average of the images.
A normal size window will give us the correct implementation and blending.


Answer to Q3:
Here again, as we geting higher and higher in the pyramid, we get better
images. Once again, since the levels are BPF, the lowest level will give
us very bright image with poor blending, since we still have most of the
details of each image.
when we get to the next levels more and more tiny details are desapire
due to the LPF, and we get better blending.
```

```
=== Section 3.1 ===

Trying to build Gaussian pyramid...
    Passed!
Checking Gaussian pyramid type and structure...
    Passed!
Trying to build Laplacian pyramid...
    Passed!
Checking Laplacian pyramid type and structure...
    Passed!

=== Section 3.2 ===

Trying to build Laplacian pyramid...
    Passed!
Trying to reconstruct image from pyramid... (we are not checking for quality!)
    Passed!
Checking reconstructed image type and structure...
    Passed!

=== Section 3.3 ===

Trying to build Gaussian pyramid...
    Passed!
Trying to render pyramid to image...
    Passed!
Checking structure of returned image...
    Passed!
Trying to display image... (if DISPLAY env var not set, assumes running w/o screen)
    Passed!

=== Section 4 ===

Trying to blend two images... (we are not checking the quality!)
    Passed!
Checking size of blended image...
    Passed!
Tring to call blending_example1()...
    Passed!
Checking types of returned results...
    Passed!
Tring to call blending_example2()...
    Passed!
Checking types of returned results...
    Passed!

=== All tests have passed ===
=== Pre-submission script done ===


    Please go over the output and verify that there are no failures/warnings.
    Remember that this script tested only some basic technical aspects of your implementation
    It is your responsibility to make sure your results are actually correct and not only
    technically valid.
```

2 README

```
mottig
sol3.py
answer_q1.txt
answer_q2.txt
answer_q3.txt
```

3 answer q1.txt

```
As we learn in class, laplacian pyramids can use as a band-pass-filter.
By multiplying each level with some coefficient, we actually applying the BPF.
For example, multiplying the smallest level by 0 will filter the low frequencies,
in another words - LPF. multiplying the biggest one by zero will give us HPF. and any
other level and coefficients combining a BPF.
```

4 answer q2.txt

```
Here is another frequencies filtering. For too small window we let all the details remain in
both images, which cause us to see all the edges as we let the high frequencies stay, so the
blending is of course not good enough.
For too big window, on the other hand, we creating a strict low pass filter, which will also
cause us to get only very blur details, like the average of the images.
A normal size window will give us the correct implementation and blending.
```

5 answer q3.txt

```
Here again, as we geting higher and higher in the pyramid, we get better
images. Once again, since the levels are BPF, the lowest level will give
us very bright image with poor blending, since we still have most of the
details of each image.
when we get to the next levels more and more tiny details are desapire
due to the LPF, and we get better blending.
```

6 sol3.py

```python
import numpy as np
from scipy import signal as sp_signal
from scipy.ndimage import filters
from scipy.misc import imread
from skimage.color import rgb2gray
import matplotlib.pyplot as plt
import os

GREYSCALE, COLOR, RGBDIM = 1, 2, 3
MAX_PIX_VAL = 255
PYR_IDX = 0  # the index of pyr in the tuple returned by build_gaussian_pyramid function
MIN_SIZE = 16  # minimum size of an image
SAMPLE_FACTOR = 2  # determine down/up sampling frequency - e.g. when reducing image take one of each 2 pixels



def is_valid_args(filename: str, representation: int) -> bool:
    """
    Basic checks on the functions input
    """
    return (filename is not None) and \
           (representation == 1 or representation == 2) and \
           isinstance(filename, str)


def read_image(filename: str, representation: int) -> np.ndarray:
    """
    Reads a given image file and converts it into a given representation
    :param filename: string containing the image filename to read.
    :param representation: representation code, either 1 or 2 defining if the output should be either a
    greyscale image (1) or an RGB image (2)
    :return: Image represented by a matrix of class np.float32, normalized to the range [0, 1].
    """

    if not is_valid_args(filename, representation):
        raise Exception("Please provide valid filename and representation code")

    try:
        im = imread(filename)
    except OSError:
        raise Exception("Filename should be valid image filename")

    if im.ndim == RGBDIM and (representation == GREYSCALE):  # change rgb to greyscale
        return rgb2gray(im).astype(np.float32)

    elif im.ndim != RGBDIM and (representation == COLOR):
        raise Exception("Converting greyscale to RGB is not supported")

    return im.astype(np.float32) / MAX_PIX_VAL


def gaussian_kernel(size: int) -> np.ndarray:
    """
    create a gaussian kernel
    :param size: the size of the gaussian kernel in each dimension (an odd integer)
    :return: gaussian kernel as np.ndarray contains np.float32
    """
    if not size % 2: # if size is even number
        raise Exception("kernel size must be odd number")
```

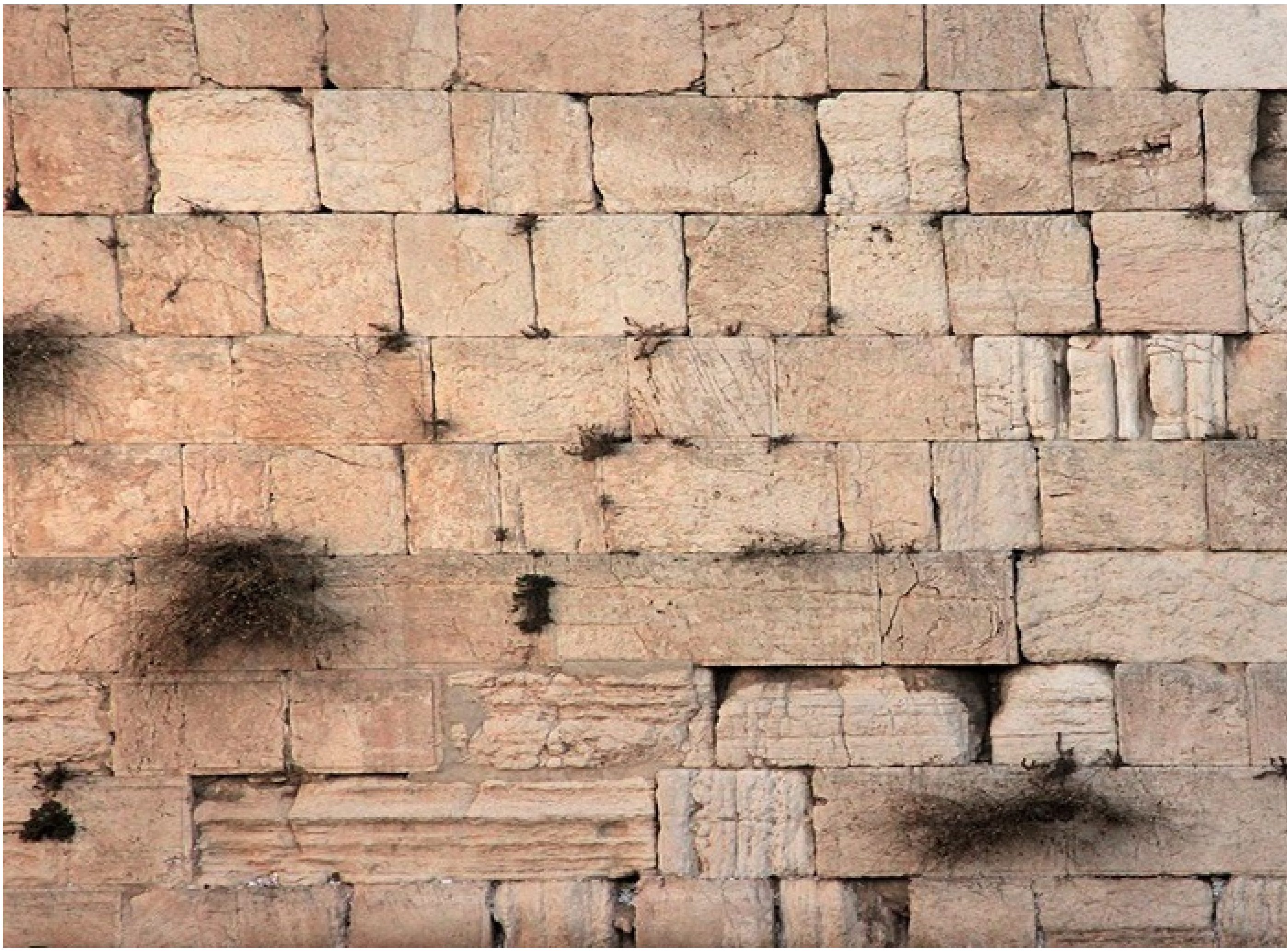
```python

    base = np.ones((1, 2), np.float32)  # gaussian kernel base - [1 1] vector
    kernel = base
    for i in range(size-2):
        kernel = sp_signal.convolve(kernel, base)
    kernel /= np.sum(kernel)  # normalize kernel
    return kernel


def reduce(im: np.ndarray, blur_filter: np.ndarray) -> np.ndarray:
    """
    reduce an image by blurring and reducing image size by half
    :param im: image to reduce
    :param blur_filter: filter to use for blurring
    :return: the reduced image
    """
    reduced_im = filters.convolve(im, blur_filter, mode='mirror')
    reduced_im = filters.convolve(reduced_im, blur_filter.T, mode='mirror')
    reduced_im = reduced_im[::SAMPLE_FACTOR, ::SAMPLE_FACTOR]  # take any second element of im (if SAMPLE_FACTOR==2)
    return reduced_im


def expend(im: np.ndarray, blur_filter: np.ndarray) -> np.ndarray:
    """
    expend an image by doubling the image size and then blurring
    :param im: image to expend
    :param blur_filter: filter to use for blurring
    :return: the expended image
    """
    expended_im = np.zeros(([SAMPLE_FACTOR*dim for dim in im.shape]), dtype=np.float32)
    expended_im[1::SAMPLE_FACTOR, 1::SAMPLE_FACTOR] = im  # zero padding each odd index (if SAMPLE_FACTOR==2)
    doubled_filter = 2 * blur_filter
    expended_im = filters.convolve(expended_im, doubled_filter, mode='mirror')
    expended_im = filters.convolve(expended_im, doubled_filter.T, mode='mirror')
    return expended_im


def build_gaussian_pyramid(im: np.ndarray, max_levels: int, filter_size: int) -> (list, np.ndarray):
    """
    construct a Gaussian pyramid of a given image
    :param im: a grayscale image with double values in [0, 1].
    :param max_levels: the maximal number of levels in the resulting pyramid.
    :param filter_size: the size of the Gaussian filter (an odd scalar that represents a squared filter)
    :return: tuple contains list of the pyramid levels and the filter used to construct the pyramid
    """
    filter_vec = gaussian_kernel(filter_size)
    pyr = [im]
    for lvl in range(1, max_levels):
        if min(pyr[-1].shape) <= MIN_SIZE:
            break
        pyr.append(reduce(pyr[-1], filter_vec))
    return pyr, filter_vec


def build_laplacian_pyramid(im: np.ndarray, max_levels: int, filter_size: int) -> (list, np.ndarray):
    """
    Construct a Laplacian pyramid of a given image
    :param im: a grayscale image with double values in [0, 1].
    :param max_levels: the maximal number of levels in the resulting pyramid.
    :param filter_size: the size of the Gaussian filter (an odd scalar that represents a squared filter)
    :return: tuple contains list of the pyramid levels and the filter used to construct the pyramid
    """
    gauss_pyr, filter_vec = build_gaussian_pyramid(im, max_levels, filter_size)
    pyr = [gauss_pyr[i] - expend(gauss_pyr[i+1], filter_vec) for i in range(len(gauss_pyr)-1)]
    pyr.append(gauss_pyr[-1])  # add G_n level as is
    return pyr, filter_vec
```

```python
def laplacian_to_image(lpyr: list, filter_vec: np.ndarray, coeff: np.ndarray) -> np.ndarray:
    """
    :param lpyr: list of laplacian pyramid images
    :param filter_vec: the filter used to create the pyramid
    :param coeff: vector of coefficient numbers to multiply each level
    :return: the reconstructed image as np.ndarray
    """
    im = lpyr[-1]*coeff[-1]
    for i in range(len(lpyr)-1, 0, -1):
        im = expand(im, filter_vec) + lpyr[i-1]*coeff[i-1]
    return im


def render_pyramid(pyr: list, levels: int) -> np.ndarray:
    """
    render a given pyramid
    :param pyr: a Gaussian or Laplacian pyramid
    :param levels: the number of levels to present in the result  max_levels.
    :return: black image in which the pyramid levels of the given pyr are stacked horizontally
    """
    levels = levels if levels <= len(pyr) else len(pyr)
    res = np.zeros((pyr[0].shape[0], sum([pyr[i].shape[1] for i in range(levels)])), np.float32)
    border = 0
    for lvl in range(levels):
        min_pix, max_pix = np.min(pyr[lvl]), np.max(pyr[lvl])
        cur_im = (pyr[lvl] - min_pix) / (max_pix - min_pix)
        res[0:cur_im.shape[0], border:border+cur_im.shape[1]] = cur_im
        border += cur_im.shape[1]
    return res


def display_pyramid(pyr: list, levels: int) -> None:
    """
    display the rendering of a given pyramid
    :param pyr: a Gaussian or Laplacian pyramid
    :param levels: the number of levels to present in the result  max_levels.
    """
    plt.figure()
    plt.imshow(render_pyramid(pyr, levels), cmap=plt.cm.gray)
    plt.show()


def pyramid_blending(im1: np.ndarray, im2: np.ndarray, mask: np.ndarray,
                     max_levels: int, filter_size_im: int, filter_size_mask: int) -> np.ndarray:
    """
    pyramid blending as described in the lecture
    :param im1: first grayscale image to be blended
    :param im2: second grayscale image to be blended
    :param mask: boolean  mask representing which parts of im1 and im2 should appear in the resulting im_blend
    :param max_levels: the maximal number of levels to use in the pyramids.
    :param filter_size_im: size of the Gaussian filter used in the construction of the pyramids of im1 and im2.
    :param filter_size_mask: size of the Gaussian filter used in the construction of the pyramid of the mask.
    :return: the blended image as np.ndarray
    """
    if im1.shape != im2.shape != mask.shape:
        raise Exception("im1, im2 and mask must agree on dimensions")

    l1, filter_vec = build_laplacian_pyramid(im1, max_levels, filter_size_im)
    l2 = build_laplacian_pyramid(im2, max_levels, filter_size_im)[PYR_IDX]
    g_m = build_gaussian_pyramid(mask.astype(np.float32), max_levels, filter_size_mask)[PYR_IDX]
    l_out = [g_m[k]*l1[k] + (1 - g_m[k])*l2[k] for k in range(len(l1))]
    im_blend = laplacian_to_image(l_out, filter_vec, np.ones(len(l_out), np.float32)).clip(0, 1)
    return im_blend


def get_blending_images(im1_path: str, im2_path: str, mask_path: str) -> tuple:
    """
    helper function for examples functions - prepare and return all needed images
```

```python
    :param im1_path: path of im1 to blend
    :param im2_path: path of im2 to blend
    :param mask_path: path of mask
    :return: im1, im2, mask (bool array), im_blend_template (the template for the blended image)
    """
    dir_path = os.path.dirname(__file__)
    im1 = read_image(os.path.join(dir_path, im1_path), 2)
    im2 = read_image(os.path.join(dir_path, im2_path), 2)
    mask = read_image(os.path.join(dir_path, mask_path), 1)
    mask[mask <= 0.1] = 0
    mask[mask > 0.1] = 1
    mask = mask.astype(np.bool)
    im_blend_template = np.zeros(im1.shape, np.float32)
    return im1, im2, mask, im_blend_template


def display_example(im1: np.ndarray, im2: np.ndarray, mask: np.ndarray, blended: np.ndarray,) -> None:
    """
    helper function to display the examples
    """
    f = plt.figure()
    f.add_subplot('221', title='im1')
    plt.imshow(im1, cmap=plt.cm.gray)
    f.add_subplot('222', title='im2')
    plt.imshow(im2, cmap=plt.cm.gray)
    f.add_subplot('223', title='mask')
    plt.imshow(mask, cmap=plt.cm.gray)
    f.add_subplot('224', title='im_blend')
    plt.imshow(blended, cmap=plt.cm.gray)
    plt.show()


def blending_example1() -> tuple:
    """
    blend image of The Dude with image of Albert Einstein using pyramid_blending
    :return: the blended image
    """
    im1, im2, mask, im_blend = get_blending_images('imgs/d.jpg', 'imgs/e.jpg', 'imgs/dude_mask.jpg')
    for clr_idx in range(RGBDIM):
        im_blend[:, :, clr_idx] = pyramid_blending(im1[:, :, clr_idx], im2[:, :, clr_idx], mask, 7, 35, 55)
    display_example(im1, im2, mask, im_blend)

    return im1, im2, mask, im_blend


def blending_example2() -> tuple:
    """
    blend image of The Western Wall with image of people studying at the Aquarium, using pyramid_blending
    :return: im1, im2, mask, im_blend
    """
    im1, im2, mask, im_blend = get_blending_images('imgs/c.jpg', 'imgs/aq.jpg', 'imgs/mask_aq.jpg')
    for clr_idx in range(RGBDIM):
        im_blend[:, :, clr_idx] = pyramid_blending(im1[:, :, clr_idx], im2[:, :, clr_idx], mask, 7, 35, 15)
    display_example(im1, im2, mask, im_blend)

    return im1, im2, mask, im_blend
```

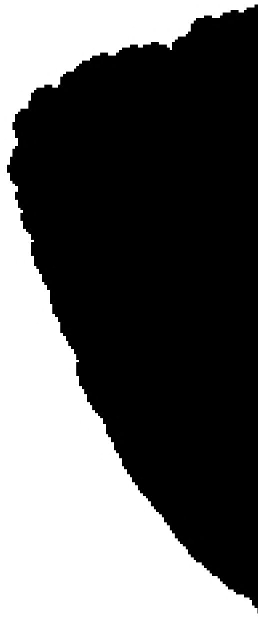
7 imgs/aq.jpg

8 imgs/c.jpg

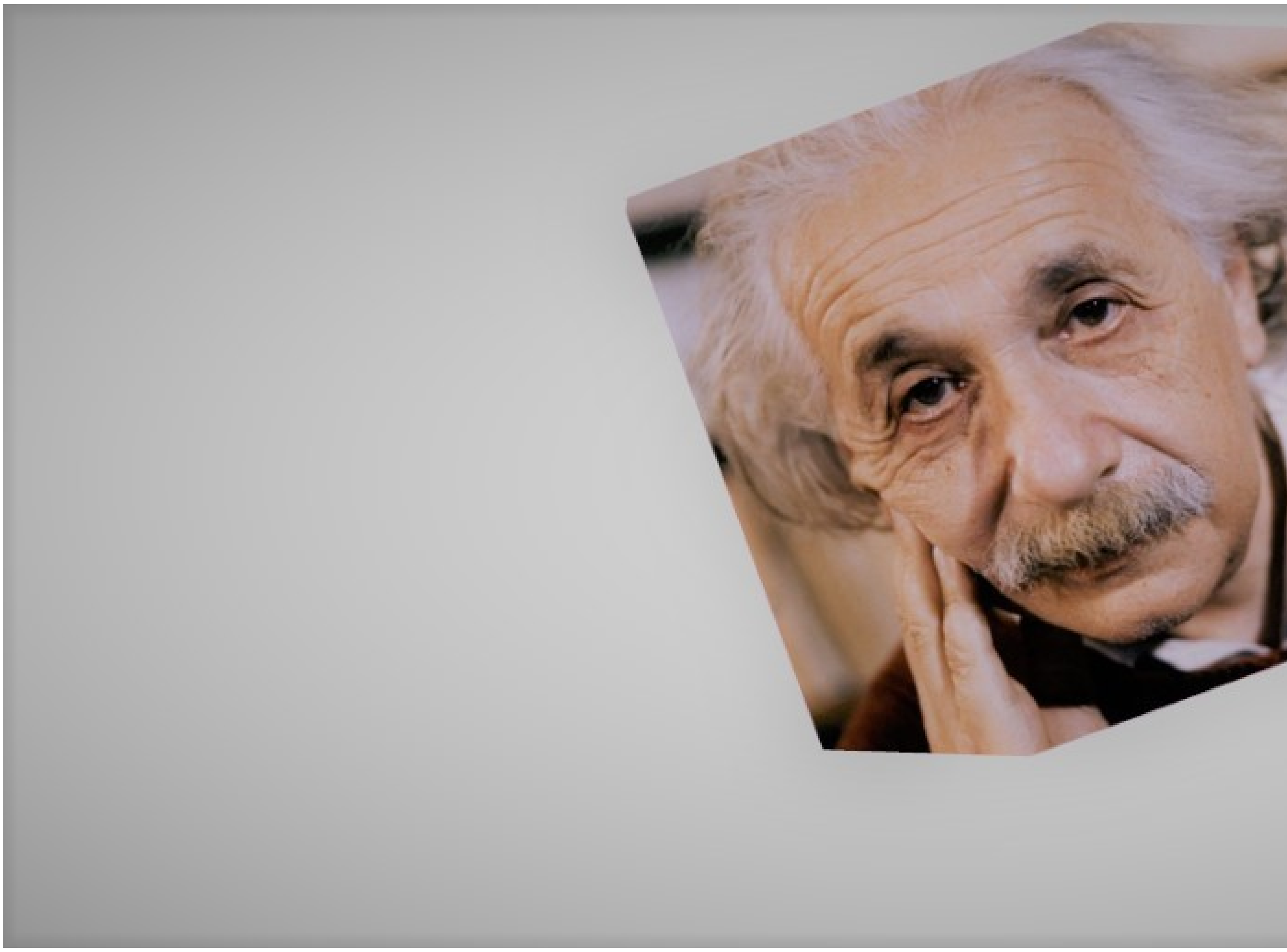


9 imgs/d.jpg

10 imgs/dude mask.jpg



11 imgs/e.jpg



12 imgs/mask aq.jpg