Assignment 3: Simple Image Segmentation and Compositing

Guilherme Gomes Haetinger - 00274702

Proposal:
The goal of this assignment is to familiarize the students with notions of image segmentation
and image compositing. For this assignment, you will play with image segmentation using the Intelligent
Scissors and the GrabCut algorithms, and with
the use of Laplacian sequences and alpha compositing.

Image Segmentation

Inteligent Scissors

Gimp's inteligent scissor is very intuitive. I applied it to these two Grêmio legends and here are the results:

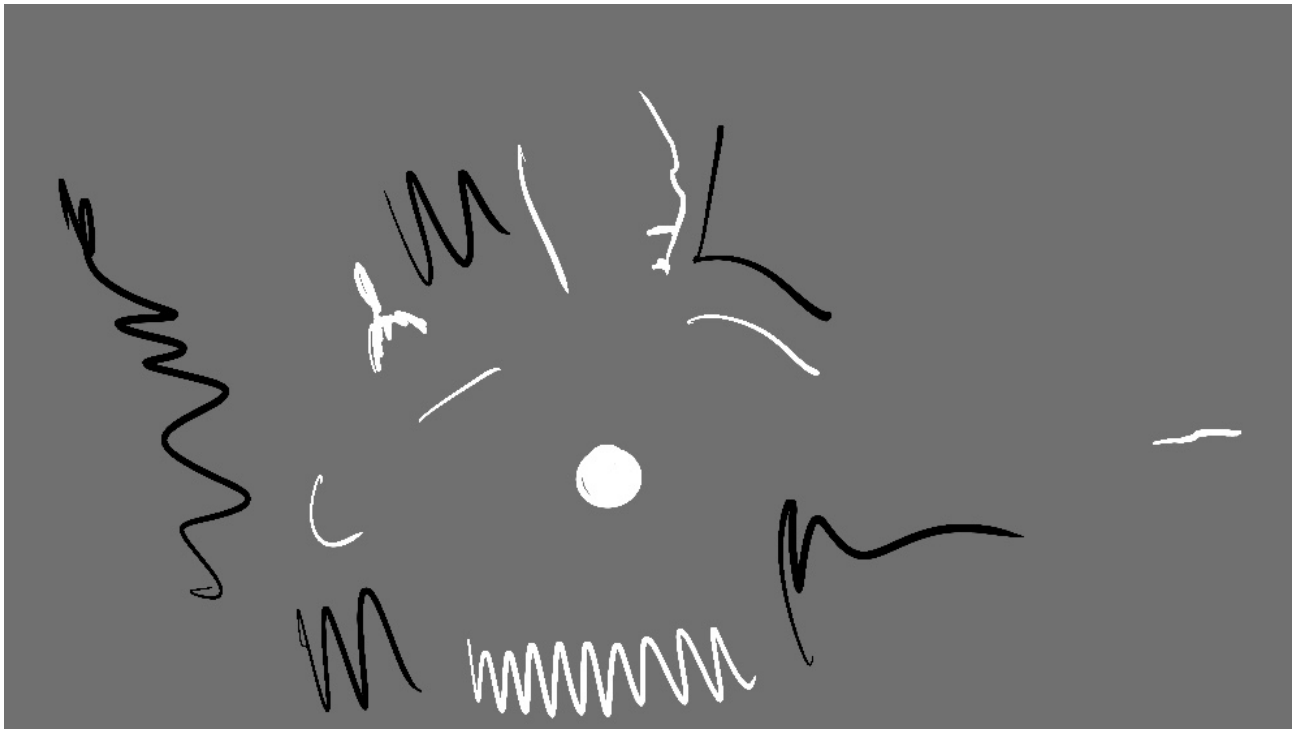
Scribbles

Using Scribbles to segment a picture is also straightforward. Using my python script *"create_segment.py"* which specializes a mask by creating a rectangle bound and checking pixel similarities, and then uses a scribble I drew to introduce user based foreground/background probability and generate better results.

```
@pyinclude("./create_segment.py")
```

Result of rectangle bound segmentation

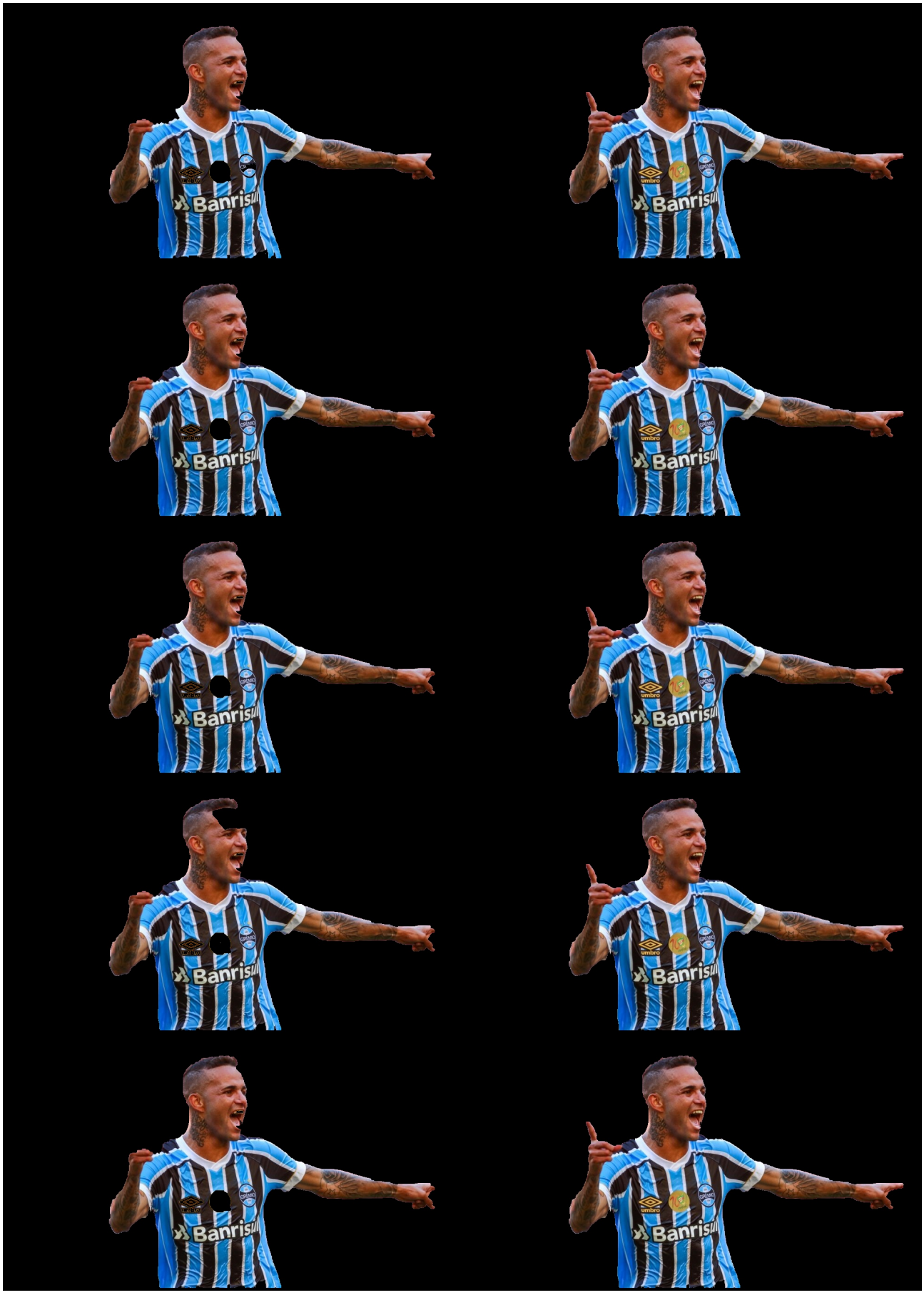
My scribbles



Result of scribble results


```
iters = [1, 5, 10, 15, 20]
```

One of the parameters we could work with, was number of iterations on the mask approximation process. More iterations equals more precision. Below, I display (left) the rectangle segmentation result and (right) the scribble process result. The results increase number of iterations from top to bottom $[1, 5, 10, 15, 20]$.


Banrisul
umbro

From these results, we don't see much of an improvement of result with the increased number of iterations. Zooming in, we might be able to see the edges of the shirt become more smooth though.

Laplacian Compositing

Gaussian Sequences

```
apple = load("./res/Apple.png");
```

```
orange = load("./res/Orange.png");
```

Our objective in this session is to create the proper image sequences to build a mixed picture of both images above.

To start this off, we create a function that generates a *Gaussian Sequence* for any given image. For this, aside of the image, I set two different parameters:

- $\Delta\sigma$

  Step to which the σ used for the *Gaussian filters* increases over the image sequence;

- seq_size

  Number of filters/images generated for the sequence;

Both of these alter how frequencies that are filtered out on the image sequence.

```
make_gaussian_sequence (generic function with 1 method)
```

```
function make_gaussian_sequence(I, Δσ, seq_size)
    filters = [Kernel.gaussian([s, s], [51, 51]) for s ∈ collect(Δσ:Δσ:
    (seq_size-1)*Δσ)];
    return vcat([I], [conv(collect(filters[i]), I)[26:end-25, 26:end-25]
    for i ∈ 1:length(filters)])
end
```

Below, we have the resulting *Gaussian Sequences* for both images using $\Delta\sigma = 2, \text{seq\_size} = 5$

(a vector displayed as a row to save space)

```
make_gaussian_sequence(apple, 2, 5)
```

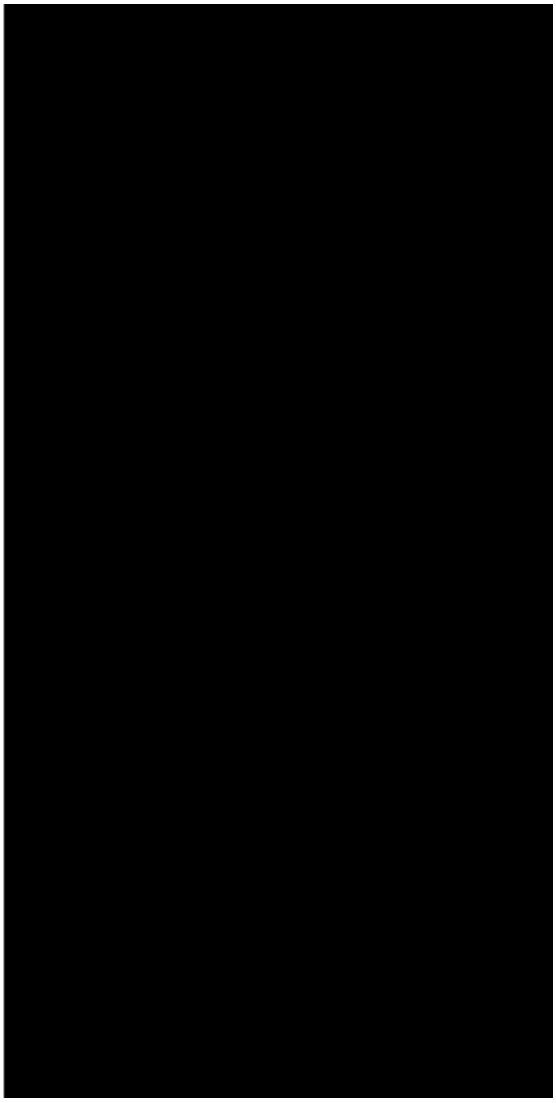
(a vector displayed as a row to save space)

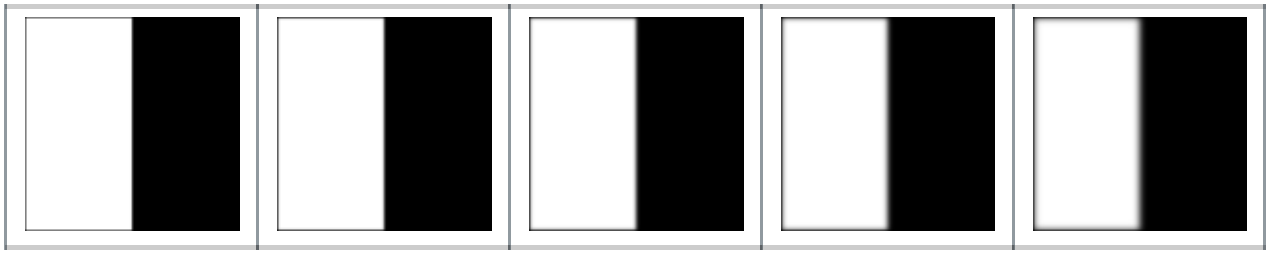
```
make_gaussian_sequence(orange, 2, 5)
```

Mask

Now that we have the images' *Gaussian Sequences*, we're only missing one: the mask's. First, we define a very simple mask and then apply the same function:

```
mask =
```





(a vector displayed as a row to save space)

```
make_gaussian_sequence(mask, 2, 5)
```

Laplacian Sequences

Generating a *Laplacian Sequence* is simple once we have the respective *Gaussian Sequences* layed out. To do so, we simply copy the sequence and do the following operation until $i = |G_s| - 1$:

$$L_s[i] = L_s[i] - G_s[i+1]$$

where L_s means *Laplacian Sequence* and G_s means *Gaussian Sequence*.

```
md"""
Generating a *Laplacian Sequence* is simple once we have the respective
*Gaussian Sequences* layed out. To do so, we simply copy the sequence and
do the following operation until $i = |G_s|-1$:

$L_s[i] = L_s[i] - G_s[i+1]$

where $L_s$ means *Laplacian Sequence* and $G_s$ means *Gaussian
Sequence*.
"""
```

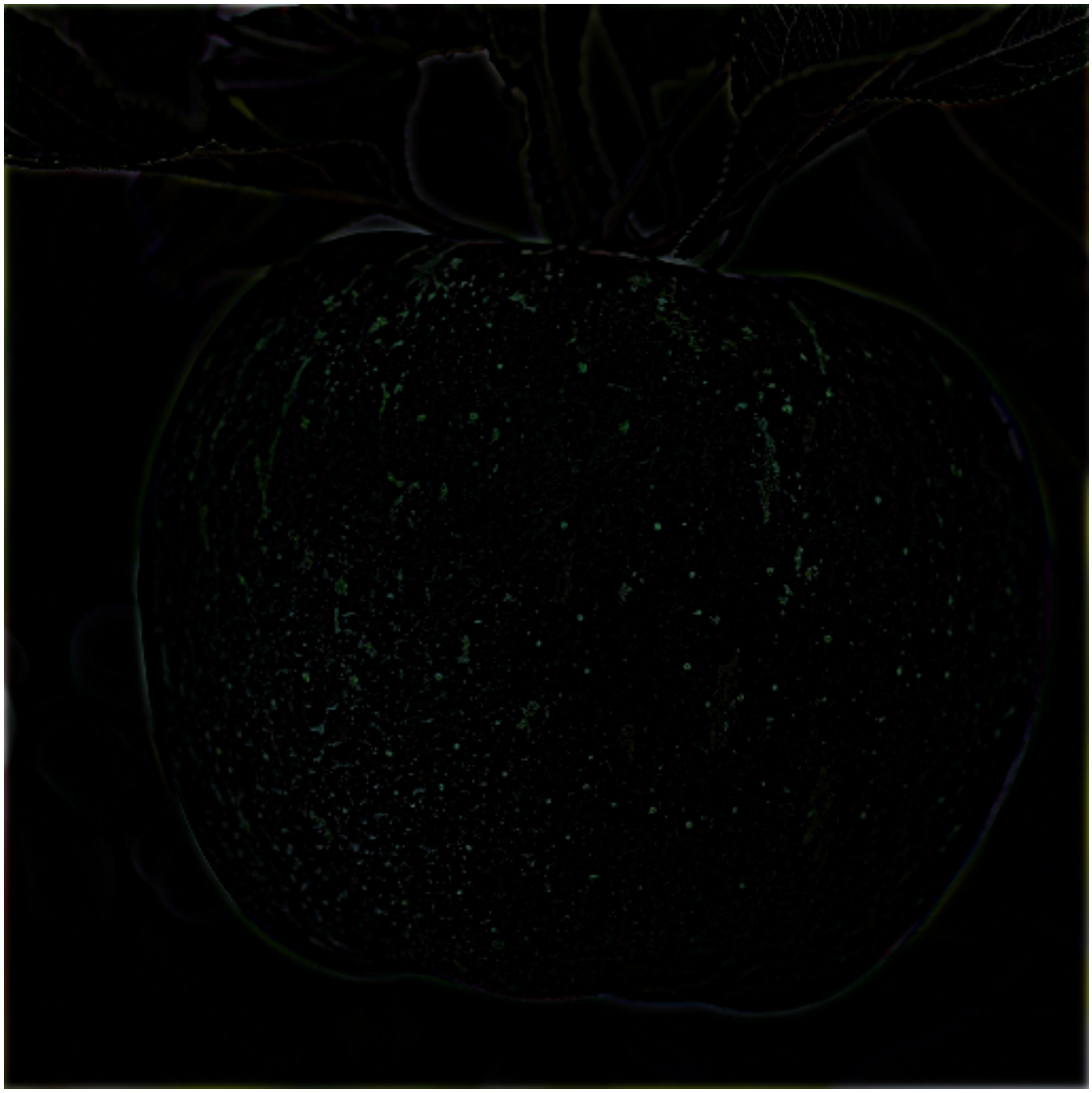
```
make_laplacian_sequence (generic function with 1 method)
```

```
function make_laplacian_sequence(gaussian_sequence)
    laplacian = copy(gaussian_sequence)
    for (i, gauss) ∈ enumerate(gaussian_sequence[2:end])
        laplacian[i] = laplacian[i] .- gauss
    end
    return laplacian
end
```

Example:

(a vector displayed as a row to save space)

```
make_laplacian_sequence(make_gaussian_sequence(apple, 2, 5))
```



We see that the sharp images are very dark because the smooth color transitions are filtered out! Zooming in like in the image above, we can see the high frequencies of the image.

Blending

Finally, we can blend two sequences using a mask by following this equation:

$$\sum_{i=1}^{N} M_i * A_i + (1 - M_i) * B_i$$

which is what we do in the followint function.

```
blend_two_lapl_sequences (generic function with 1 method)
```

```
function blend_two_lapl_sequences(A, B, M)
    M_inv = [1 .- m for m ∈ M]
    return sum([
        M[i] .* A[i] + M_inv[i] .* B[i]
        for i ∈ (1:length(A))
    ])
end
```

```
join_images (generic function with 1 method)
```

```
function join_images(A, B, M, Δσ, seq_size)
    A_lapl = make_laplacian_sequence(make_gaussian_sequence(A, Δσ,
    seq_size))
    B_lapl = make_laplacian_sequence(make_gaussian_sequence(B, Δσ,
    seq_size))
    M_gauss = make_gaussian_sequence(M .|> Gray, Δσ, seq_size) .|> (m ->
    Float64.(m))
    blend_two_lapl_sequences(A_lapl, B_lapl, M_gauss)
end
```

Below, I explore changing the variables to understand what would be the best combination.

$$\Delta\sigma = 2, \text{sequence size} = [1, 5, 10, 100], \text{respecitvely}$$

$$\text{sequence size} = 10, \Delta\sigma = [1, 2, 5, 10], \text{respecitvely}$$

It seems very clear that a larger sequence size and Δσ generate better results. However, using $\text{seq\_size} = 100$ takes around *10s* to run, which I don't think is a good amount of time to wait. Viewing the images above, I thought the best result was to use $\Delta\sigma = 5, \text{seq\_size} = 10$, the extra examples proved me wrong and showed the best results used $\Delta\sigma = 2, \text{seq\_size} = 10$.

Other examples

First example: Tom Hank's freaky hands

(a vector displayed as a row to save space)

```
join_images(hankeye, hank, hankmask, 2, 10)
```

Second Example: Pizza or Books? That's the question...

(a vector displayed as a row to save space)


```
join_images(pisa, berkeley, pisamask, 2, 10)
```

Alpha Compositing

Starting off the Alpha Compositing section, I need to load the three images and transform the alpha image into a *floating-point* matrix.

```
background = load("./res/background.png");
```

```
Png warn: iCCP: known incorrect sRGB profile
```

```
alpha = load("./res/GT04_alpha.png");
```

```
Png warn: iCCP: known incorrect sRGB profile
```

```
foreground = load("./res/GT04.png");
```

```
alpha_float = alpha .|> Gray .|> Float64;
```

After being able to run the composting equation and putting our little friends into the background set for the assignment, they went for a few adventures:

- Went to eat some oranges in Pelotas, RS
- Played against Inter in Gauchão and beat them by 10x0, each of them scoring a hat-trick
- Decided to go study at UC Berkely, where they graduated in EECS with honors

```
alpha_float .* foreground + (1 .- alpha_float) .* background
```

Shown in the codeblock above, we see that Julia makes it very easy for us to use the compositing equation.

Zooming in on the picure, however, we see there are a few artifacts on the hair of our two little friends, e. g. the left side of the red hair shows tones of green, which are clear leftovers of the original image, where they were placed in a green field. This has to be because of how hard it is to define the opacity of these very small details in the image.

Extra Picture

Below, we have a picture of me and Mao Tse Tung at the SF MOMA. It looks very boring and so our colorful buddies want to fix it by being a part of it!

```
mao =
```

To do so, we need to do the following:

1. Create a blank canvas with the resolution of my picture;
2. Scale and rotate the foreground image;
3. Fill a set of coordinates with the foreground pictures;
4. Repeat 2. and 3. for the alpha mask image;

```
(newforeground, newalpha) = let
    newforeground = zeros(RGB{N0f8}, size(mao))
    scaledforeground = imresize(foreground, floor.(Int64, size(foreground)
    .* 3.3))
    rotated_img = imrotate(scaledforeground, -π/100)
    (height, width) = size(rotated_img)
    newforeground[1500:1499+height, 800:799+width] = rotated_img

    newalpha = zeros(RGB{N0f8}, size(mao))
    scaledalpha = imresize(alpha, floor.(Int64, size(alpha) .* 3.3))
    rotated_alpha = imrotate(scaledalpha, -π/100)
    newalpha[1500:1499+height, 800:799+width] = rotated_alpha
    (newforeground, newalpha .|> Gray .|> Float64)
end;
```

Finally, we get the following, much more fun, image:

```
newalpha .* newforeground + (1 .- newalpha) .* mao
```

Conclusion

This was a very fun assignment which I didn't have nearly as much time to do as the past ones. Nonetheless, it was very simple and so I didn't need as much time. I think the pictures that were achieved by this look absolutely great and this shows how simple algorithms can be very useful and generate outstanding results. Needless to say I learnt a lot.