POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Deep learning 3D facial reconstruction framework for prosthetic rehabilitation

Supervisors

Prof. Stefano DI CARLO

Prof. Alessandro SAVINO

Prof. Roberta BARDINI

Candidate

Jose VILLALOBOS

March 2022

Summary

Aesthetics has been one of the concepts most studied and analyzed by the greatest philosophers, authors and thinkers of all times and is nowadays a key factor in case of prosthetic rehabilitation, especially when connected to facial rehabilitation (e.g., in case of severe accidents or pathologies).

The implementation of the aesthetic standards in daily clinical practice to improve the aesthetics of patients has been a constant challenge for clinicians since dentistry was born. The traditional approach in this domain is to create wax models or similar artifacts that can help the patient to visualize the result if the prosthetic rehabilitation process. The advance in computing graphics techniques, machine learning and artificial intelligence has the potential to significantly impact this domain.

In this project we explore the advantages of face reconstruction into the field, and draw a path between 2D images and a conditioned 3D face model. The end-to-end process includes pre-processing the image files, converting to 3D, merging two models, and replacing a region with a third source. For each step, this report addresses the main characteristics and requirements, discuss about the fundamentals, and provides concept examples to illustrate the results. On the background, we use programming language to write new tools, review the theory on computer graphics and get a shallow understanding on the implementation of the reconstructor.

Acknowledgements

Vorrei ringraziare Dio, per spingermi forte sempre avanti, e fissare in questa vicenda una piccola stella nei tempi di buio, soltanto per ricordarci che più oscuro diventa, meglio brilla la sua costellazione. A te tutto questo successo...

Table of Contents

List of Figures

Acronyms

AI
: artificial intelligence

ML
: machine learning

2D
: two dimensions

3D
: three dimensions

BFM
: basel face model

CPU
: central processing unit

GPU
: graphical processing unit

MTCNN
: multi-task cascaded convolutional neural network

CV
: computer vision

TF
: tensor flow

Chapter 1

Introduction

Deep learning 3D facial reconstruction framework for prosthetic rehabilitation is a research collaboration project between the Politecnico di Torino and Universita' degli Studi di Torino, intended to explore, study and develop tools to help doctors and patients during a prosthetic rehabilitation by improving the graphical visualization of the procedure and expected results even before the actual intervention.

The target application is facial analysis, as great advances on digital fields like computer vision and face recognition create a suitable path for this scope, opening opportunities to work with the main face regions such as forehead, eyes, nose and dental injuries.

Providing doctors with a 3D model of the face of the patient represents a powerful tool having positive effects on diagnose, enhanced planning and execution, risk assessment, and also better understanding from the recipient that can follow along the process in a more intuitive approach.

Throughout this process, from patient to renovated 3D model, several steps are taken. From getting the face scanned into a digital representation, analyze the model by detecting the damaged areas, modify such model following the changes introduced by the medical intervention, to replacing areas with healthy donor models, among others. To help boost this pipeline, Deep learning techniques can be incorporated, speeding up data clustering and analysis, as well as faster reconstruction model.

We then focus on the early stages of this long path, dig into the 3D model reconstruction from simple 2D photos, review the fundamentals and variables present on the conversion, inspect different methods to optimize the model, and review the ties this reconstruction model has with the other disciplines involved at a higher project scale.

The 3D face reconstruction tool utilized, as detailed in the next sections, requires one single picture to generate the model. As part of the project requirements, detailed information from the two sides of the face (i.e, left and right) are of interest. To achieve this, two separate models are generated independently for each side and later combined in the 3D format. After merging, the model goes through a match and replace algorithm to swap a given area with the same region from another model, intended to be a healthy donor model. Figure 1.1 shows the scope of this project. In the next sections, we'll start from the reconstruction details, what additional information has to been extracted from it in order to accurate merge the left and side models, and how to perform a damaged part replacement.

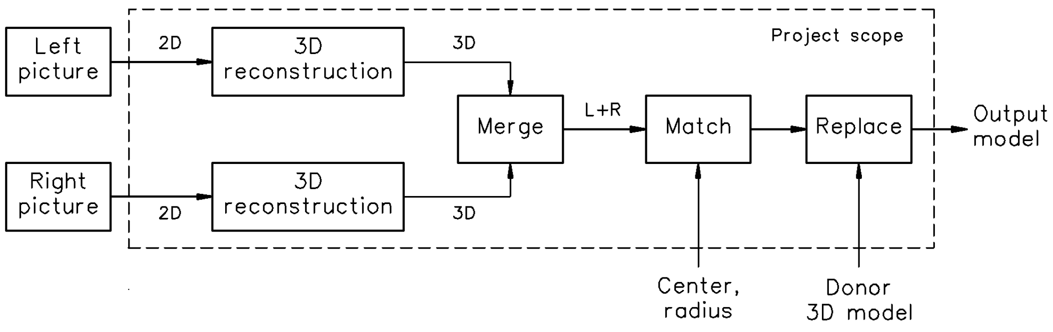


Figure 1.1: Project scope

1.1 Objectives

1. Reconstruct left/right facial 3D models starting from paired 2D photos taken with commercial low cost cameras.
2. Extract the rotation parameters of the models.
3. Study the output 3D model file format.
4. Merge the left and right models into an unified model.
5. Match and replace a desired area with a secondary 3D model.
6. Discuss on advantages, limitations and future work.

1.2 Previous work

The thesis report in [1] introduces the concept of *aesthetics*, meaning and history of the word. Discusses the intrinsic value of beauty in nature and humankind, and correlates the value into the odontology field. From this perspective, the document develops the theory behind the geometry and symmetry of the human face, covering front and lateral views to finally dig into the dental area. The author then focuses on the detailed region of the mouth, presents the importance in context between the nose and chin, and how the lips and teeth play the role on the smile along with the parameters implied.

On the second half, the document addresses the importance of utilizing digital photos to improve and speed analysis in dental procedures, recalls the evolution of 3D models that help creating resin molds, and introduces different techniques to obtain an object's 3D model from 2D images. Regarding the process to convert 2D face images into a 3D model, several approaches and algorithms are compared from which fast implementations using Machine Learning and Neural Networks stand out. The author also provides a brief on these topics and the tools used on the demonstration examples. From this last family of single image face modelers, the author proposes using a face reconstruction model owned by Microsoft and explains why it fits better than other implementations for the proposed application. The tool generates a 3D model from a single image using weakly-supervised learning, boasts high accuracy and texture fidelity while being easy to use, fast and robust. In addition, the output model is aligned with the source picture for easy overlay.

1.3 Computer environment

A low-range (4 core, 8GB, specs below) computer running Debian GNU/Linux is used for the project. This allow for the installation of the required packages, edition and execution of the scripts (programs) and visualize the results. Aside from basic software tools that ship with most distributions, only `python` is required, `git` is recommended.

Processors: 4 x AMD A12-9720P RADEON R7, 12 COMPUTE CORES 4C+8G
Memory: 7.2 GiB of RAM
Graphics Processor: AMD Radeon R7 Graphics

To setup the environment, read and follow the steps described in [2] to install the face reconstruction generator. This repository is also a good source of information and references to the processes involved during training and reconstruction.

Have a look at the *Testing requirements* section and notice some features, namely training and rendering, are only available on GNU/Linux. We will use the `conda` framework to handle the `python` environment, `TensorFlow version 1.12` (no gpu version) installed using `pip`, this allows using the pre-compiled binary file `rasterize_triangles_kernel.so`, otherwise follow the instructions to compile the `tf_mesh_renderer` before continuing. For visualization will be using the *MeshLab* application. On Appendix A a list of package versions used during the development of this thesis project are provided.

After setting up the environment, downloading all the required files and doing the instructed file moves and edits described on the repository in [2], you should have a folder structure similar to the one provided in Appendix B.

1.4 Image file format

Image files are very common and widely spread nowadays, popular formats for example are BMP, JPEG, JPG, TIFF, SVG and PNG. The face reconstruction tool used in Section 2.2 can work with `.jpg` and `.png` files, while the pre-processing stage in Section 2.1 can also handle `.jpeg`. However, to simplify the set-up is recommended to convert all images to `.jpg` as a first step. You can easily convert between image file formats using the `convert` tool from the *ImageMagick Studio* suite.

1.5 Input files

Each subject supplies a paired (left and right) photos with an isometric (≈45°) view of the head, see Figure 1.2. Across the execution of the project two datasets were provided, with the main difference that in the first batch people stand with the mouth closed, while in the second smiling.

(a) Right side **(b)** Left side

Image taken from Iconfinder under CC BY-ND 2.5 DK license.

Figure 1.2: Input files

Datasets included paired pictures for 10 and 15 adult persons respectively, taken on white-is backgrounds, portrait oriented and clearly taken with a regular (phone) camera. Image resolutions varying from 768x1024px to 1538x2048px. File sizes span from 70 to 550 KiB each. File formats correspond to all .jpeg the first dataset, while the second to .jpg. Filenames follow enumeration starting from 1, adding the suffix "-left" or "-right", for example '8-left.jpeg'.

Chapter 2

Model Reconstruction

This chapter is focused on examining the face reconstruction process, overview the components and fundamentals behind, and provide details on the steps taken to generate the model.

2.1 Image pre-processing

In order to feed the 3D model reconstruction tool, the image needs to be pre-processed to extract the main five landmarks of the face, these are the two eyes, nose and mouth boundaries. Documentation in [2] recommends using `dlib` or `mtcnn`, we will use the latter which can be downloaded from its repository or installed via `pip` as instructed in [3].

The Multi-task Cascaded Convolutional Neural Network (MTCNN) is a face detector implementation written in Python by Iván de Paz Centeno, based on the original FaceNet's MTCNN implementation from David Sandberg, which is further based on the paper from Zhang, K et al (2016)[1]. This technique implements face detection with pose estimation, considering illumination and occlusions, and exploiting deep convolutional neural networks to produce a robust and fast method suitable to unconstrained environments.

The file *preproc.py* in Section 7.2 is a slightly modified version of the original *example.py* in [3] that iterates over all files in an *input* folder. The tool uses TensorFlow for the detector computation, and OpenCV for handling and editing the image. In case the TF version conflicts with the one used on the next steps, a simple way to workaround this is to create a different `conda` environment and install the desired package version.

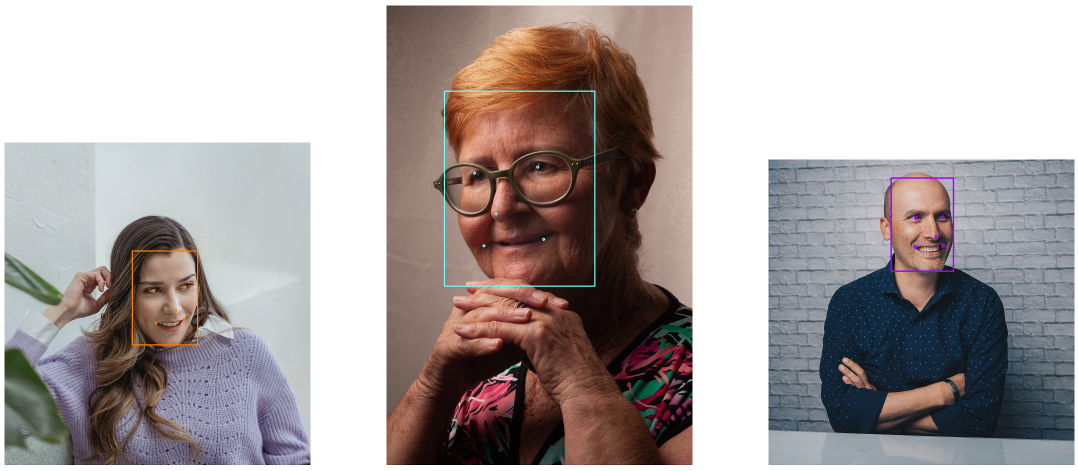
[1]See references [3], [4], [5]

For each input image file, the results are written into the *output* folder, these are a newly created image with the graphical result and a the text file containing the five *(x,y)* landmark coordinates. One can also examine the output of the program and see the five landmarks information. Figure 2.1 shows an example of result of the face detection algorithm.

The next sections will require the original image and the landmark coordinates files to have the same filename.

Landmark information

```
[ {     'box': [244, 358, 390, 567],
        'confidence': 0.9999512434005737,
        'keypoints': {
            'left_eye': (296, 583),
            'right_eye': (470, 578),
            'nose': (352, 709),
            'mouth_left': (303, 781),
            'mouth_right': (483, 779)
        }   } ]
```



Pictures taken from Pexels courtesy of George Milton, Marcelo Verfe and Apunto Group.

Figure 2.1: Image pre-processing, face landmark detection

2.2 Face Reconstructor

To run the face modeler, we use the script `demo.py` provided in [2]. After moving the original image files, thus not the "_mtcnn.jpg" files created in the previous step, along with the corresponding `.txt` landmark files into the folder *Deep3DFaceReconstruction/input* within the Microsoft engine. The script processes all files in this directory and writes the results into the *output* folder

By inspecting the code run during the 3D modeling, one can layout the main steps taken during computation, Figure 2.2 shows a top-level sequential execution diagram, where the first step is loading the 2D images in memory. Afterwards, a standard face is created by loading a morphable 3D Basel Facel model (BFM)[6] and the corresponding five landmarks of interest, which are calculated from a bigger set of 68 similarity landmarks provided with the model. The landmarks have the form *(x,y,z)* and the model contains $m = 53490$ vertices, this last property is addressed on Section 2.3.

Consequently, the face decoder is implemented using a network *graph* that contains a Face Reconstruction model, which handles variables such as face shape, identity, expression, texture, skin and landmarks. The starting weights can be loaded using the pre-trained saved model file provided `network/FaceReconModel.pb` or a *checkpoint* (`.ckpt`) file that one can generate by training the model with a custom dataset (see Section 2.3).

The reconstructor model includes auxiliary information that helps to characterize the face and the environment, such as camera, color, illumination, position and rotation. The latter is required to perform the *merge* step in Chapter 4, therefore, the next section presents the modifications introduced to extract this information from the model graph.

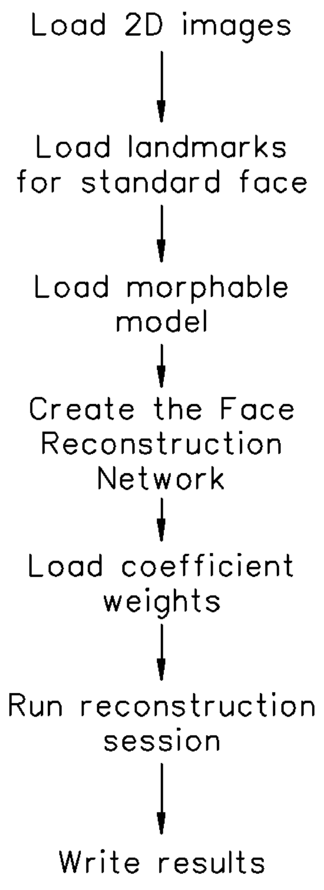
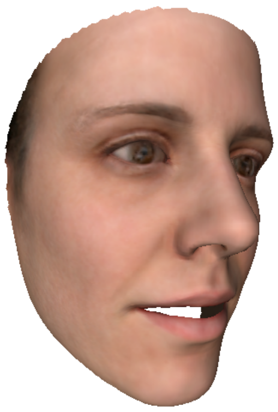
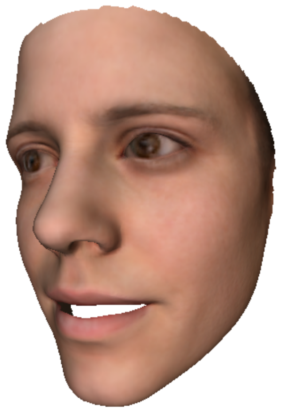
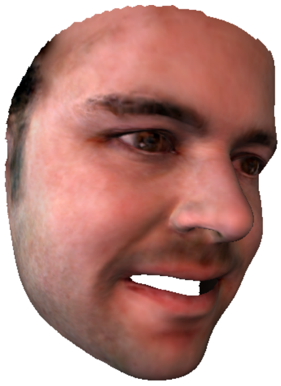

Load 2D images
Load landmarks for standard face
Load morphable model
Create the Face Reconstruction Network
Load coefficient weights
Run reconstruction session
Write results

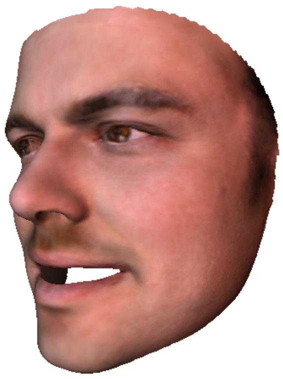
Figure 2.2: Face reconstruction model sequence

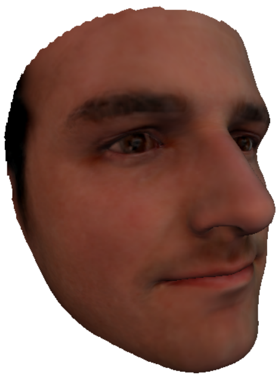
A *session* runs the reconstructor to shape the morphable model and fit each of the input source images. When finished, the output is the 3D model (`.obj`) and a Matlab binary data file (`.mat`) containing numerical variables of the parameters handled by the reconstructor on the face decoder, this file will contain the rotation matrix that is of our interest. Figure 2.3 shows the rendered 3D models for a left and right pair of pictures for a given subject.


(a) Sample 1. Right side


(b) Sample 1. Left side


(c) Sample 2. Right side


(d) Sample 2. Left side


(e) Sample 3. Right side

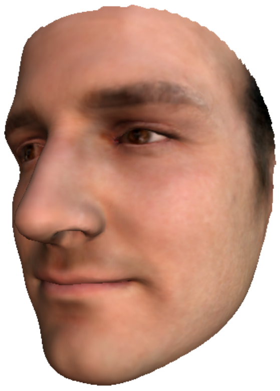

(f) Sample 3. Left side

Figure 2.3: Rendered 3D model

2.3 Training the model

The previous step used the `.pb` file as pre-trained weight values for the reconstruction. One can train the model using a specific set of pictures.

Training is available only in Linux. Note that the process demands high computer power and can take a considerable time depending upon the available resources. Documentation in [2] calls for *"Training a model with a batchsize of 16 and 200K iterations takes 20 hours on a single Tesla M40 GPU."*. To compare, when training using the hardware described in Section 1.3, the average time per iteration is 35s, making it impractical to run. One can control the training parameters in `option.py`. In addition, the tool uses a GPU device by default, the following changes are to be performed in order to use the CPU.

Differences in: `preprocess_img.py`

```
127c127
<  with tf.Graph().as_default() as graph, tf.device('/cpu:0'):
---
>  with tf.Graph().as_default() as graph, tf.device('/gpu:0'):
```

Differences in: `preprocess_img.py`

```
72c72
<  apply(prefetch_to_device('/cpu:0', None)) # When using dataset.prefetch, ...
---
>  apply(prefetch_to_device('/gpu:0', None)) # When using dataset.prefetch, ...
```

Training instructions are provided in [2]. A second pre-processing step is required prior training that takes the five landmarks and generate a bigger 68 landmark set. Similar as the procedure performed in Section 2.1, the original `.jpg` pictures and the 5-landmark `.txt` files are put into the *input* folder, but then the `preprocess_img.py` script is used and will generate the *processed_data* folder. The output contents are the cropped-to-face picture and masks in `.png` format, and the 68-landmark files in `.txt` and `bin` formats.

The training process is similar to the one described in Section 2.1 but the `train.py` script is used instead. Another network is created to handle the expressions since the original BFM model does not handle this variations. Is also interesting to note that while the BFM model has $m = 53490$ vertices, the 3D model contains $m = 35709$ and the expression basis $m = 53215$ as commented in `utils.py 52-54` in [2]. The resulted `.ckpt` file can then be used when executing the `demo.py` script as attribute.

2.4 Rotation Matrix

The rotation matrix holds the angle values that characterize the rotation of the face according to the 2D source image, this allows to later overlay and align the 3D model with the original photo. This is of our interest since in order to do a merge of two models (left and right), these must be normalized and aligned first. The following is the output `diff` to show the modifications into the `demo.py` file.

```
88a89
>  rot_mat = FaceReconstructor.rotation
104,105c105,106
<  coeff_,face_shape_,face_texture_,face_color_,landmarks_2d_,recon_img_,tri_
        = sess.run(
            [coeff,face_shape,face_texture,face_color,landmarks_2d,recon_img,tri],
            feed_dict = {images: input_img})
---
>  coeff_,face_shape_,face_texture_,face_color_,landmarks_2d_,recon_img_,tri_, rot_mat_
        = sess.run(
            [coeff,face_shape,face_texture,face_color,landmarks_2d,recon_img,tri, rot_mat],
            feed_dict = {images: input_img})
113a115
>  rot_mat_ = np.squeeze(rot_mat_,(0))
119c121
<  savemat(
        os.path.join(
            save_path,file.split(os.path.sep)[-1].replace('.png','.mat').replace('jpg','mat')
        ),
        {'cropped_img':input_img[:,:,::-1],'recon_img':recon_img_,'coeff':coeff_,
---
>  savemat(
        os.path.join(
            save_path,file.split(os.path.sep)[-1].replace('.png','.mat').replace('jpg','mat')
        ),
        {'rot_mat':rot_mat_, 'cropped_img':input_img[:,:,::-1],'recon_img':recon_img_,'coeff':coeff_,\
```

One can see that the rotation parameters are extracted from the Face reconstructor, feed and kept also during the session, squeezed and saved along with the rest of the previous parameters into the `.mat` file. A quick inspection shows that the rotation matrix is a 3x3 matrix with the rotation vectors for each of the (x, y, z) axes, for example:

$$rot\_matrix = \begin{bmatrix} 0.888812 & 0.041870 & 0.456356 \\ -0.077970 & 0.995115 & 0.060557 \\ -0.451591 & -0.089406 & 0.887734 \end{bmatrix}$$

GNU Octave can read `.mat` files in Linux, in Python can be used the Scipy library.[2]

[2]See references [7] and [8]

Chapter 3

Mesh objects

The 3D model is saved in a *mesh* object with file extension `.obj`, this chapter is intended as an overview of this format representation and introduce basic concepts about the 3D model.

3.1 Graphical elements

While a typical image uses a raster format to characterize the rendering, modern computer graphics work rather with a scene description mechanism, comprised by objects, lights and cameras. This allows to build and render a 2D (screen) image from 3D scenes, and easily change either the camera or illumination and have the effects recalculated to update the render to reflect the changes.[1]

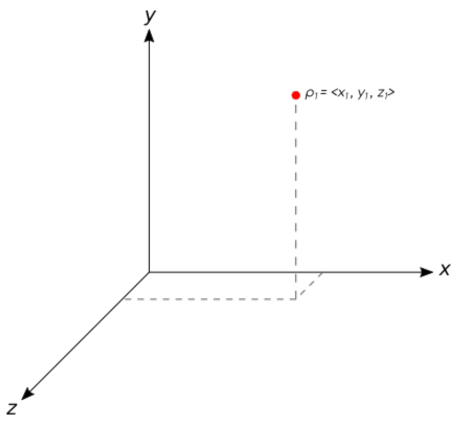
Objects are then described as a mesh, which is made using three basic primitives: points, lines and triangles.[2]

Figure 3.1a shows a red color point p_1 in a 3D space, where its position is given by the values on each (x, y, z) dimension. When having a second (blue) point p_2 (Figure 3.1b) is possible to create a line in between (Figure 3.1c). Consequently, a third (green) point p_3 creates a triangle, which is the minimum required in order to get a closed region (Figure 3.1d). To all points that land in the plane enclosed by this region are called a surface (Figure 3.1e). Therefore, the surface can be defined by using only three vertices.

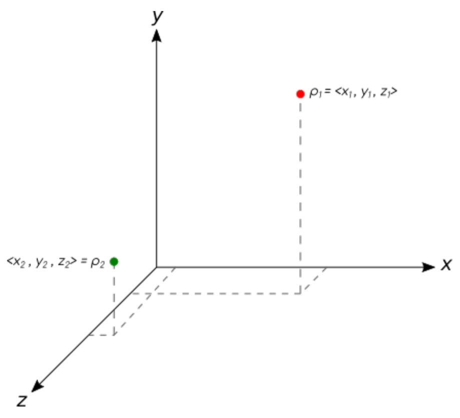
A *vertex* is a point that holds information such as position, color, texture, material and/or a normal vector. When rendering, the triangle area is filled extrapolating the three vertices' properties that build the face (Figure 3.1f).

[1]Note: A GPU is a specialized unit that performs these recalculations in a very fast and highly parallel manner.

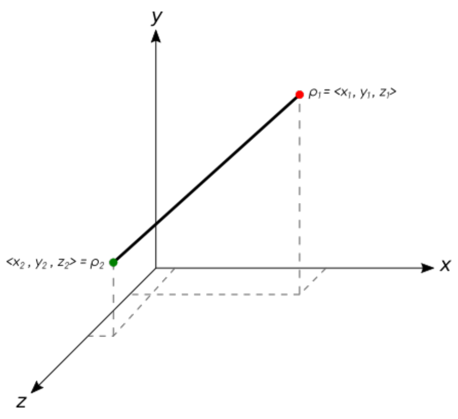
[2]Refer to [9] for a graphics library implementation documentation with a more formal definition.



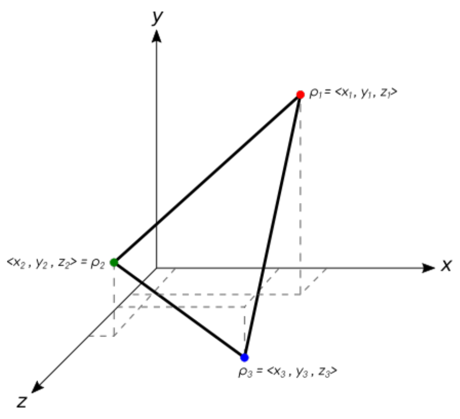
(a) A point in 3D space



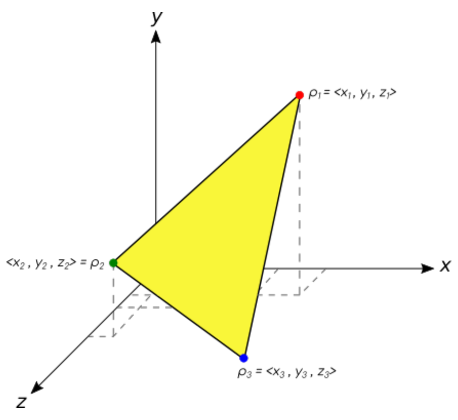
(b) A second point



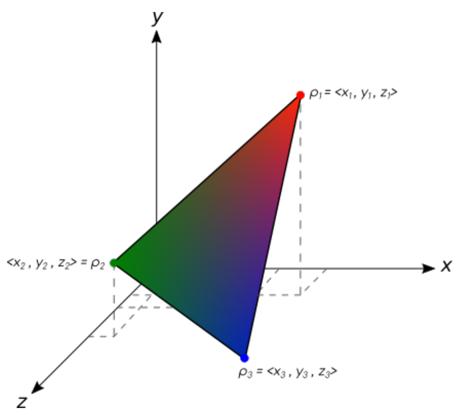
(c) A line



(d) A triangle (closed region)



(e) A surface



(f) Color texture

Figure 3.1: Graphical mesh primitives

Triangles, or faces, can be from different sizes. Adding more faces creates polygons, the mesh grows and it can then take the shape of a desired object. Figure 3.2 shows a cube made from 12 faces, or two per side. Objects can also be categorized as shells or solids (closed volumes).

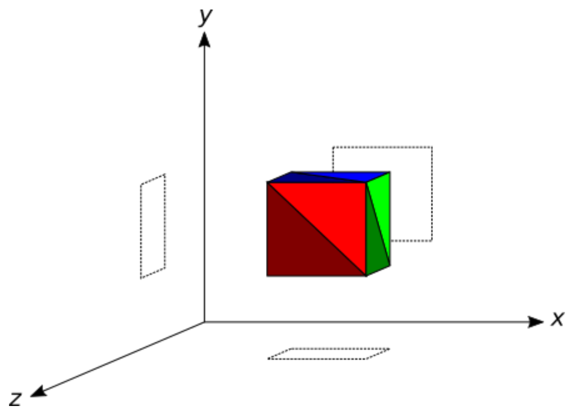


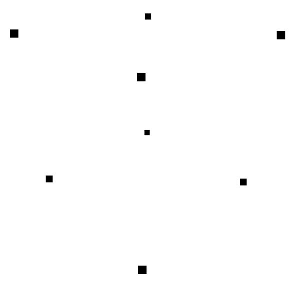
Figure 3.2: A cube and its three projections

Advanced graphic techniques include more complex methods to handle arrays of triangles and their properties, optimized given that multiple surfaces can share several points. In addition, the position of the camera and the objects define the perspective, other properties impact the way light behaves on the object, resulting in the appearance. Engine implementations use all the vertex information to enhance the render result, being able to generate transparency, shadows, reflections and smoother edges between surfaces, for example in a sphere. Most of these topics are beyond the scope of this document, we'll use position and color data per vertex (see next section) which is an actual output from the Face Reconstructor.

3.2 Face objects

Curved and detailed objects require smaller faces, and therefore a higher quantity, to cover the area and achieve a desired spatial resolution. The face models generated in the previous section contain $m = 35{,}709$ vertices and $n = 70{,}789$ surfaces. This section presents a walk through the different model perspectives, having a person's face and a cube as comparison.

Figure 3.3 shows the constellation of points in space, while in Figure 3.4 the vertices are displayed with the associated color. The triangles are then depicted in Figure 3.5, and the shape of the object is slightly uncovered. Turning the triangles solid, as shown in Figure 3.6, exposes this shape with more clarity. Finally, the model is complete when adding the color on top (Figure 3.7) and optionally applying different filters such as smoothing (Figure 3.8).


(a) Cube

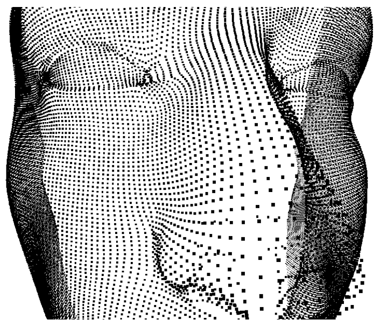
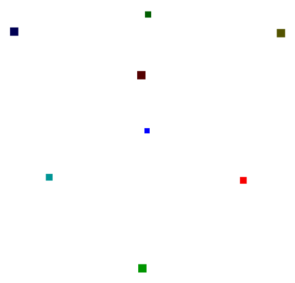

(b) Person's face

Figure 3.3: Points


(a) Cube

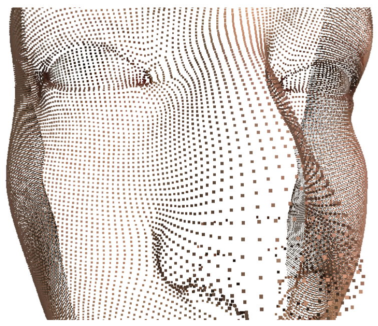
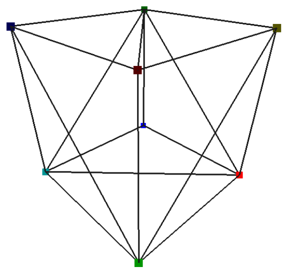

(b) Person's face

Figure 3.4: Vertices with color property


(a) Cube

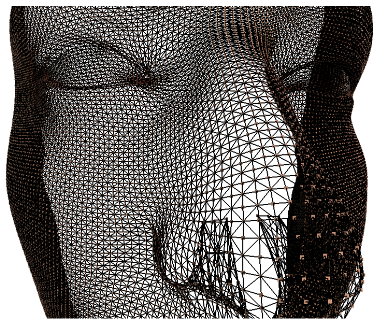
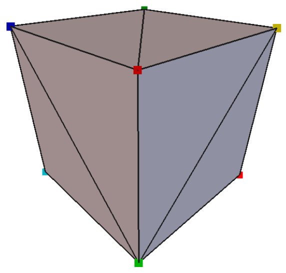

(b) Person's face

Figure 3.5: Triangles


(a) Cube

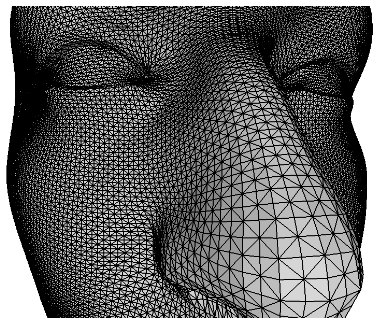
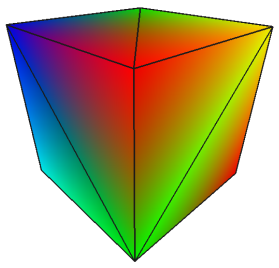

(b) Person's face

Figure 3.6: Solid


(a) Cube

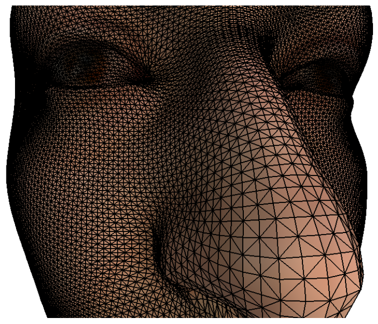
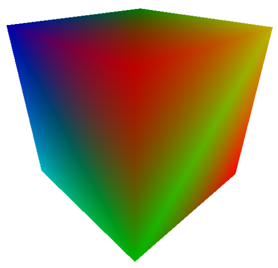
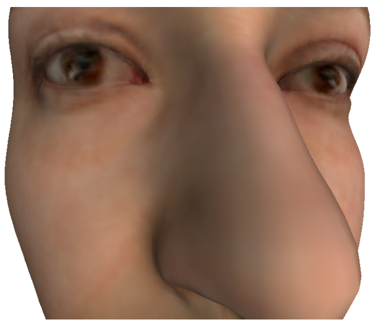

(b) Person's face

Figure 3.7: Colored model



(a) Cube



(b) Person's face

Figure 3.8: Finished model

3.3 Wavefront `.obj` file format

A mesh object file is a collection of vertices and faces.[3]

Vertices contain property values, such as $< x, y, z >$ position and $< R, G, B >$ color code, these values are usually normalized, for example $[-1{,}1]$ for position and [0,1] or [0,255] for the color. The order these vertices are declared within the file matters, since faces are declared using these indexes.

Example mesh object file

```
# Vertex lines start with the letter v
#     and are of the form v = <x,y,z,R,G,B> to include position and color.
# Face lines start with the letter f
#     and are of the form f = <v1,v2,v3>.
...
v 0.149526 -0.367906 -0.454352 0.617095 0.475823 0.352140
v 0.144221 -0.365825 -0.460239 0.615842 0.469945 0.348961
v 0.138970 -0.363530 -0.466059 0.618759 0.474185 0.351062
f 1 2 131
f 1 131 130
f 2 3 131
...
```

[3]Wavefront files accept other types of definitions, such as lines, UV textures and normals among other parameters. Refer to [10] for more information. In this project are used only the position and RGB color data per vertex.

3.4 A simple obj python class

A simple python *class* is provided in Section 7.3. The class includes generic methods for the following actions:

1. **Load**: Read an `.obj` file and load the vertex and face data into memory.
2. **Save**: Write an `.obj` file and dump the vertex and face data from memory.
3. **Limits**: Find the boundaries on all axes. With these six limits, a wrapping box can be created around the object.
4. **Translate**: Move the object by a given vector (x_0, y_0, z_0). All vertices are calculated as $< x'_i, y'_i, z'_i > = < x_i + x_0, y_i + y_0, z_i + z_0 >$
5. **Rotate**: Rotate the object on an axis by an angle. The argument is a 3 x 3 intrinsic rotation matrix on all axes. Typical rotations can be one of the x, y, z axes, or a combination of them (See Section 2.4). Vertices are recalculated as the matrix multiplication between the rotation and the original vertex position as $\overrightarrow{p'} = R \times \overrightarrow{p}$.

These methods will be useful when handling mesh objects in the next sections.

Chapter 4

Merge

After following the previous sections two independently 3D models, for left and right sides, are generated. This chapter focuses on the development of a tool in order to merge the models into a single one. Thus, the two `.obj` and `.mat` files are required to execute the steps described below.

Figure 4.1 shows the sequential steps during this process, all included in the script `merge.py` (see Section 7.4).

The algorithm starts from fetching and loading the models, later performs an alignment between models and finally the actual merging before saving the results into a new file.

The next sections present details regarding these steps. To illustrate the process, examples of paired models are provided, auxiliary colored models are included as heat maps, as well as front and isometric views to reinforce visualization throughout the process.

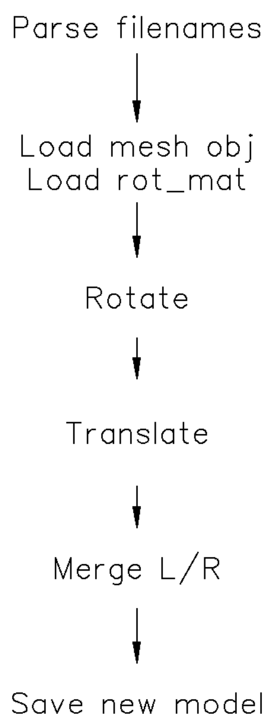


Figure 4.1: Merging sequence

4.1 Loading the files

The program takes one argument, the base of the filename to process (*bfn*), the four required files are generated internally by adding the corresponding suffixes:

1. "*bfn*-left_mesh.obj",
2. "*bfn*-right_mesh.obj",
3. "*bfn*-left.mat" and
4. "*bfn*-right.mat".

Note that the `.obj` files have conveniently inherited the "_mesh" suffix as result of the 3D conversion, these conditions can be easily changed in the script to fit other environments.

Since during the reconstruction process the two models start from a mean standard face, it can be confirmed that both models have the same quantity of vertices and faces, and that the latter are built using the same indexes on the two models. Figures 4.2 and 4.3 show an example of a paired set of models for one person.

From the model views is worth noting that the model is an open shell, that does not include hair, ears, neck or teeth details, and that the mouth is well represented whether opened or closed (see Figure 2.3. One can note that the object's position and rotation is different, adjusted to match and overlay the source photo, and can be estimated that the relative position between vertices is also different, since the reconstruction model considers expressions, and from the fact that is not exactly the same input source. The following sections evidence this effect further.

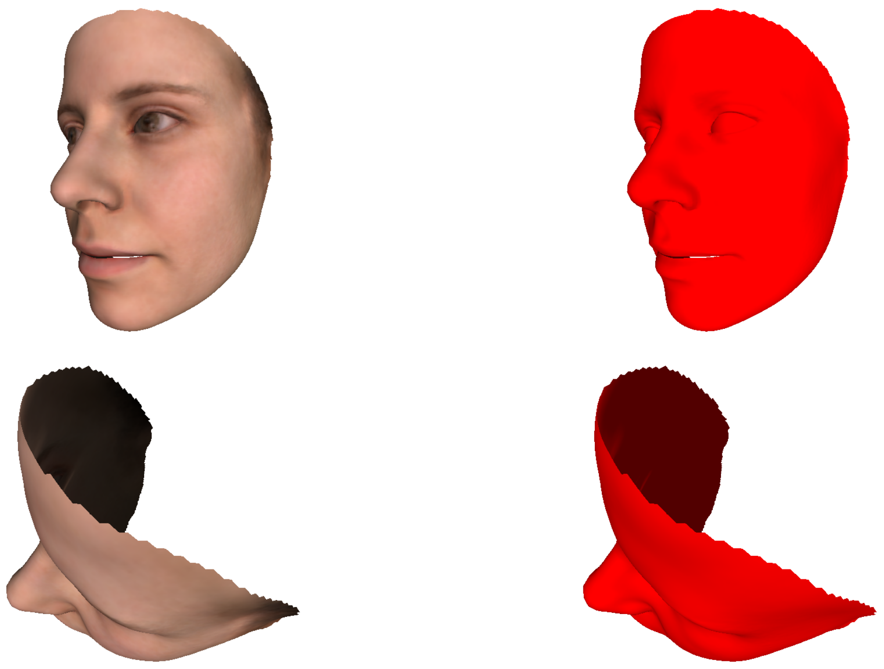


Figure 4.2: Left model

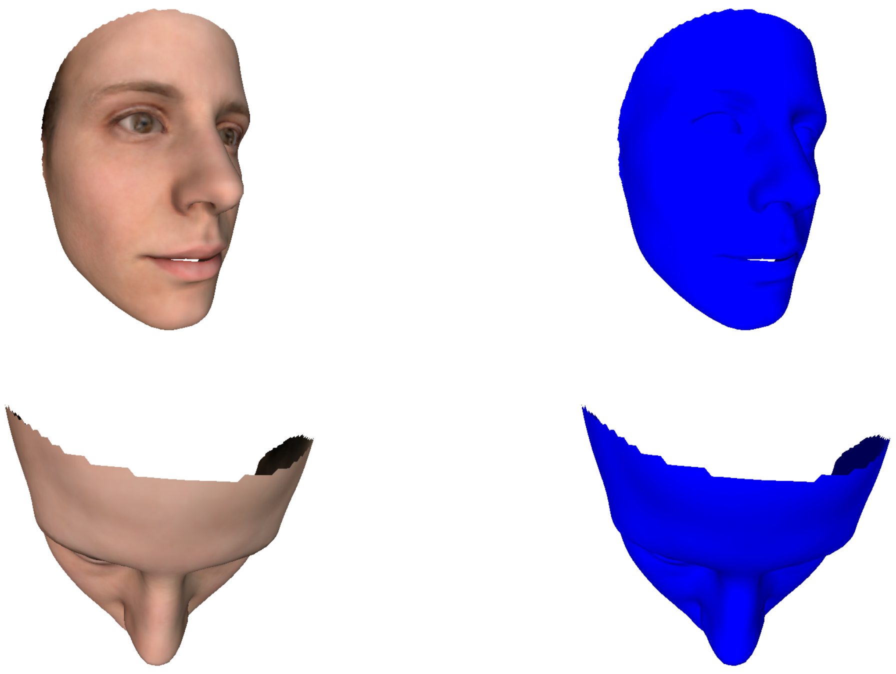


Figure 4.3: Right model

4.2 Alignment

When placing the two models under the same space, like shown in Figure 4.4, is clearly seen that each model owns a different coordinate system. To effectively merge as left and right, both models need to be aligned in order to perform the operation across the x axis, being the left the region where $x < 0$, whereas the right side complementary is defined as $x > 0$.

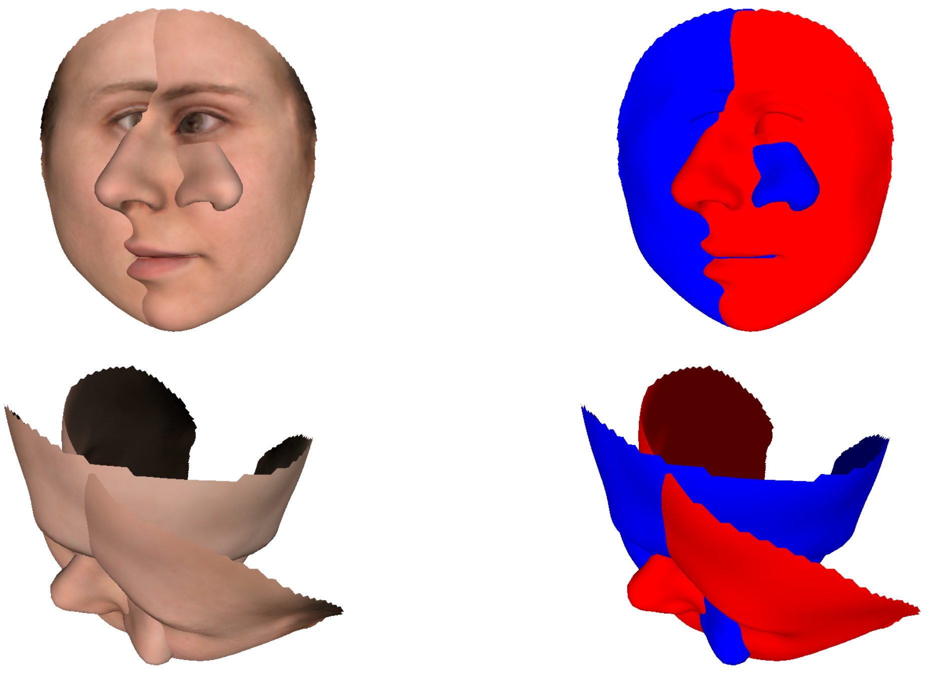


Figure 4.4: Original models

The rotation matrix, extracted from the model as described in Section 2.4, has the exact orientation along the relative coordinate system for each object. By applying the same rotation in an intrinsic manner, the object's coordinates are aligned to the overall coordinates, as shown in Figure 4.5.

However, it can be seen that this does not align both models completely. We then compute the center point of each object by calculating the boundaries on each axis and applying the corresponding offset as a translation to each vertex. The effects after the rotation and translation are shown in Figure 4.6. This peculiar overlay of the two full models demonstrate that they are actually not identical, since some regions land in front/above others (the color difference on the model and heat map view).

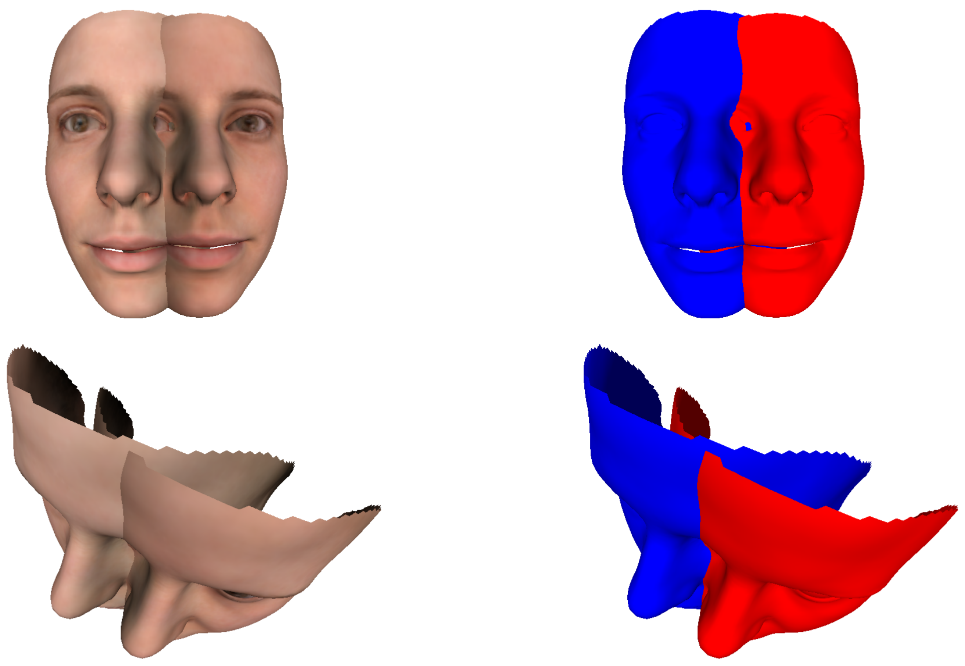


Figure 4.5: Rotated models

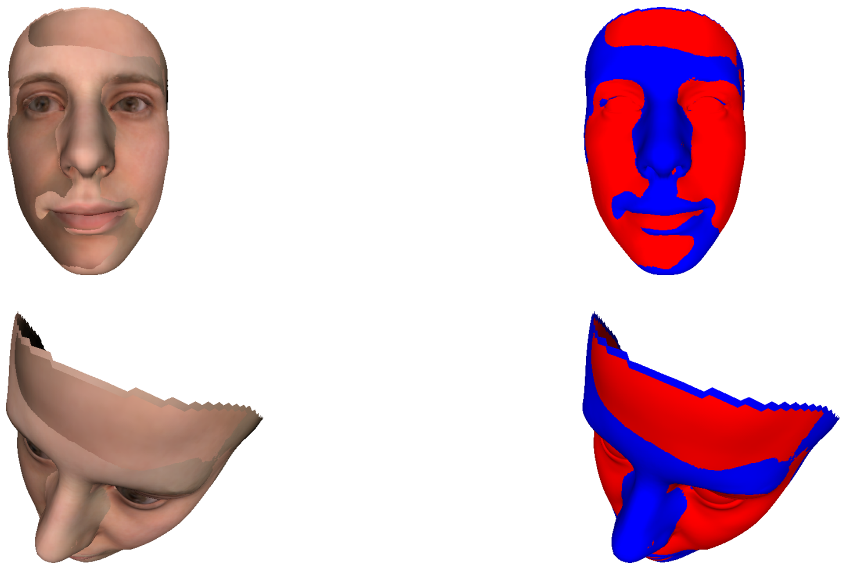


Figure 4.6: Translated models

4.3 Coarse Merge

As a first approach, let's perform a *coarse* merge between the models, this is a sharp transition at $x = 0$ as shown in Figure 4.7. Is more evident that even though both models are from the same person, with sources taken almost instantaneously, these two differ enough to mismatch considerably in position and texture, because several factors influence slightly each photo, such as illumination, angle and portion of the face shown, expressions, among others. As this approach is not the desired result, we look towards a better merging by averaging the models with some criteria.

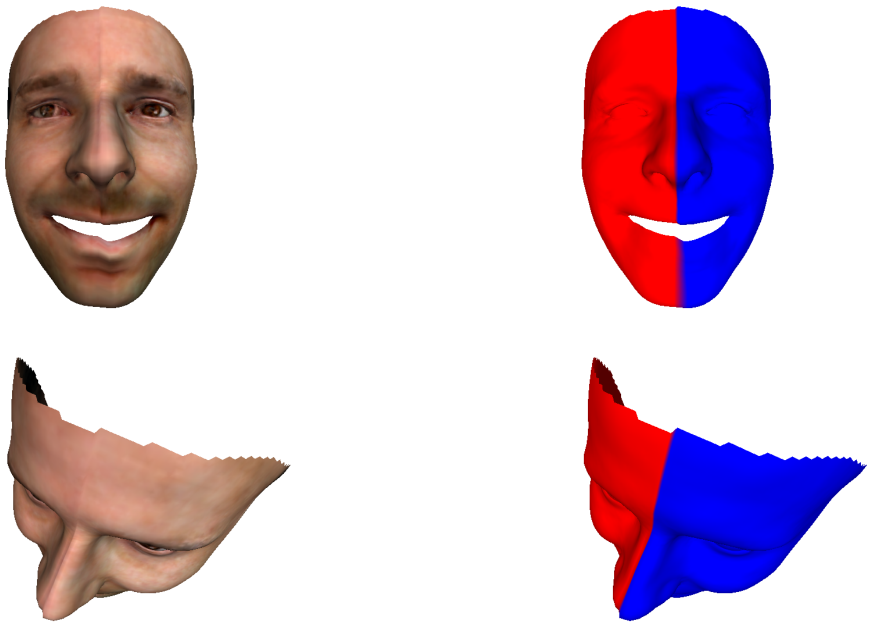


Figure 4.7: Coarse merging

4.4 Constant average merge

To average the two models a new factor γ is introduced, a value between [0,1] that defines the weight of the left model and consequently, the right model is defined as $(1-\gamma)$, the average equation is then defined as

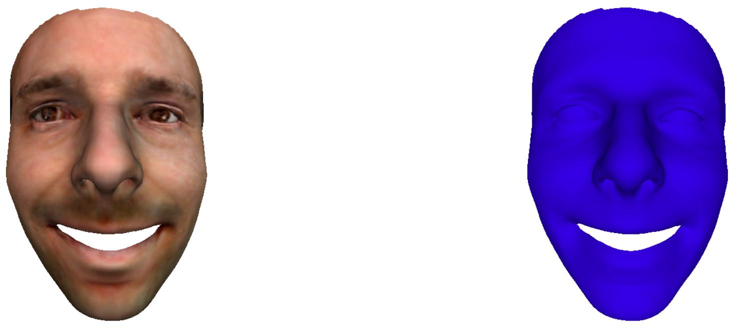
$$p_i = p_l \cdot \gamma + p_r \cdot (1-\gamma) \tag{4.1}$$

Where p_i is the new property of the vertex, and p_l, p_r are the values of the property for the left and right vertices respectively. Then for all vertices, each property (i.e, $< x, y, z, R, G, B >$) are averaged considering the two models.

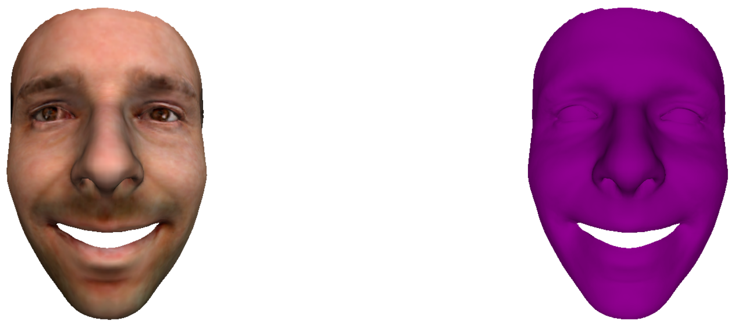
Note the effect of γ in the resulting model, when $\gamma = 0$ the model is identical to the right (blue) model, when $\gamma = 1$ the model corresponds to the left (red), and when $\gamma = 0.5$ the two models are averaged with the same proportion (purple) as $p_i = (p_l + p_r)/2$. The script *merge.py* accepts an optional *gamma* argument (default 0.5) as

```
python merge.py bfn --gamma 0.7
```

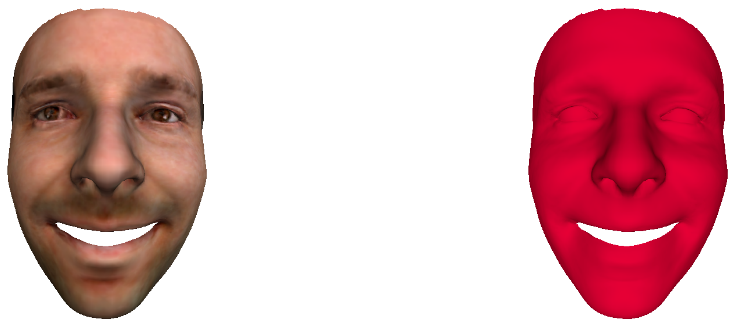
Figure 4.8 shows the resulting models for different values of γ, there are no sharp edges and the merged model is more consistent. Looking at the subtle differences on shadowing in the color models and confirmed with the heat maps, it can be seen that the average is applied homogeneously to the full object, and the color reflects the weight given to each model by the γ factor.



(a) $\gamma = 0.2$



(b) $\gamma = 0.5$



(c) $\gamma = 0.8$

Figure 4.8: Constant average merge

4.5 Linear average merge

The constant average developed above has no spatial discrimination, this is, all points in space are applied the same γ value and therefore the coloring in the heat map in Figure 4.8 is homogeneous, whether trending to red or blue, interpreted as left and right.

To emphasize and preserve the details on the lateral regions, the *left* model should weight more on the left side ($x < 0$) and vice versa, the *right* model influence more the right side of the resulted model. To achieve this, the γ value should consider the position in x, this is $\gamma = f(x)$, and using a linear interpolation we use the line equation as

$$\gamma_x = m \cdot x + b \tag{4.2}$$

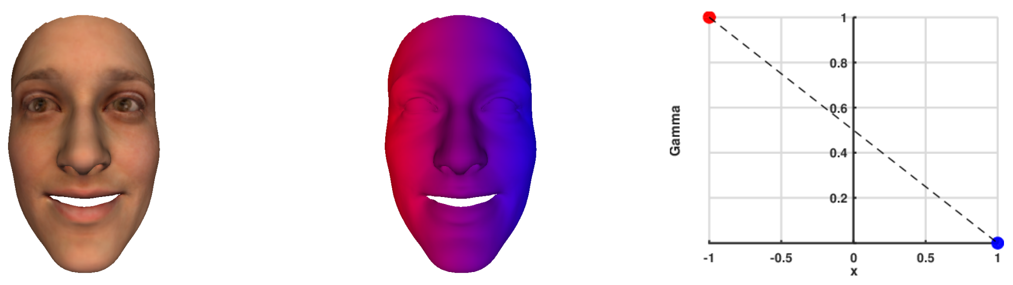
Note that the slope value $m = \Delta\gamma/\Delta x$ needs a pair of points, the normalized x values ranges from $[-1,1]$, thus $\Delta x = 2$, while we need to specify a pair of γ_l,γ_r values corresponding to the γ factor at the left-most and right-most x positions, leading to $\Delta\gamma = \gamma_r - \gamma_l$. Replacing these definitions into 4.2

$$\gamma_x = \Delta\gamma/\Delta x \cdot x + b = (\gamma_r - \gamma_l)/2 \cdot x + b \tag{4.3}$$

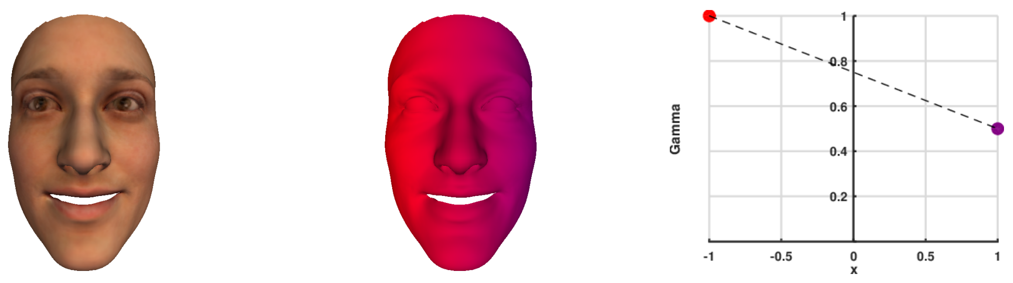
The value of b is easily derived as $b = \gamma_l + m$ using the point $(-1, \gamma_l)$.

By setting the values of γ_l and γ_r the resulting model presents a linear averaging across the x axis, Figure 4.9 shows three examples of linear averaging. The first one using $\gamma_l = 1.0$ and $\gamma_r = 0.0$, note how the left side contains mostly left model information, the opposite happens on the right side, meanwhile the center region is highly averaged. Other example configurations of γ values are included to show how one can favor the left or right models. Auxiliary graphs show the interpolation done across the x axis, the marker points reflect the color weight of γ. The script *merge.py* accepts a pair of *gamma* values separated by slash ("/"), for example to set the left $\gamma_l = 0.8$ and right $\gamma_r = 0.2$ use:

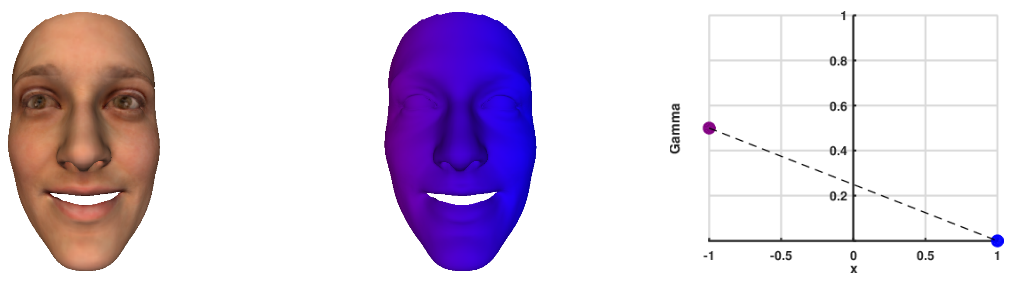
```
python merge.py bfn --gamma 0.8/0.2
```



(a) $\gamma = 1.0/0.0$



(b) $\gamma = 1.0/0.5$



(c) $\gamma = 0.5/0.0$

Figure 4.9: Linear average merge

4.6 Segmented linear merge

The linear average model uses two γ values to cover the entire x axis, we now explore a generalization of this method by adding more γ points and creating equidistant bins across the x axis. This allows for more spatial control and custom patterns.

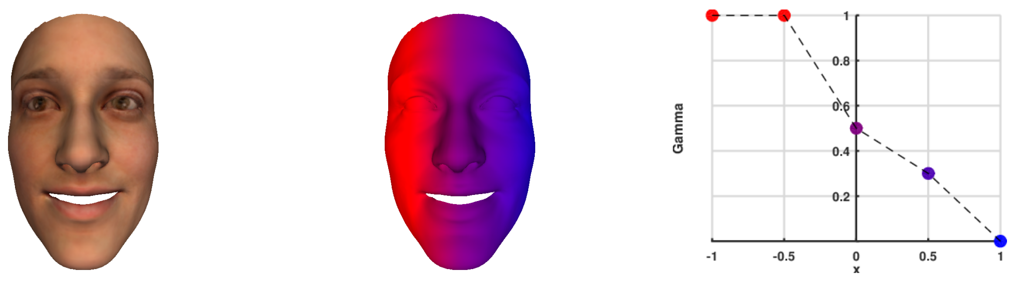
Figure 4.10 shows three particular cases, where five γ values are defined, creating 4 bins of the same size. The auxiliary graphs show the distribution of γ values versus x, and the point color represents the weight towards red or blue (left or right) models.

The script *merge.py* accepts inserting *gamma* values separated by slash ("/"), for n γ-values are created $n-1$ equally-spaced ranges automatically, for example

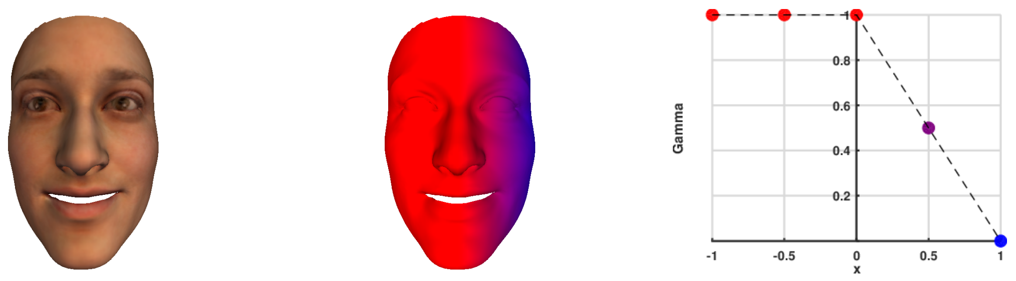
```
python merge.py bfn --gamma 1.0/0.8/0.5/0.3/0.0
```

Comments on segmented examples:

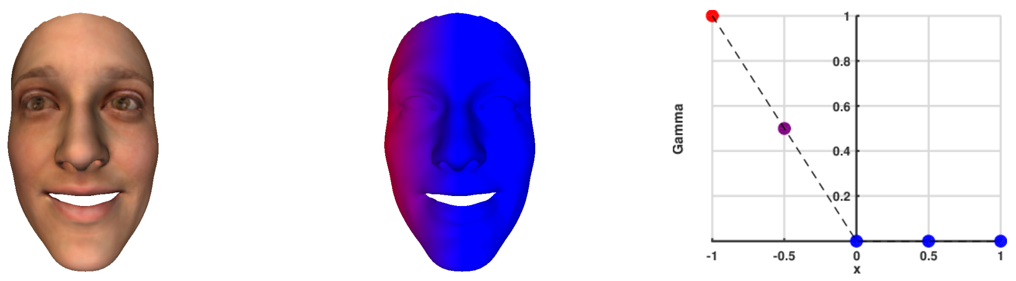
- $\gamma = 1.0/1.0/0.5/0.3/0.0$: Figure 4.10a is similar to Figure 4.9a, however it has a broader average region at the center, while the left side has been made more prominent.
- $\gamma = 1.0/1.0/1.0/0.5/0.0$: Figure 4.10b shows a different scenario from previous cases. The average region has moved towards the right, while the left side is driven by the left model.
- $\gamma = 1.0/0.5/0.0/0.0/0.0$: The counterpart of the previous case, Figure 4.10c, the resulting model trends towards the right (blue), and the average region is moved to the left side.



(a) $\gamma = 1.0/1.0/0.5/0.3/0.0$



(b) $\gamma = 1.0/1.0/1.0/0.5/0.0$

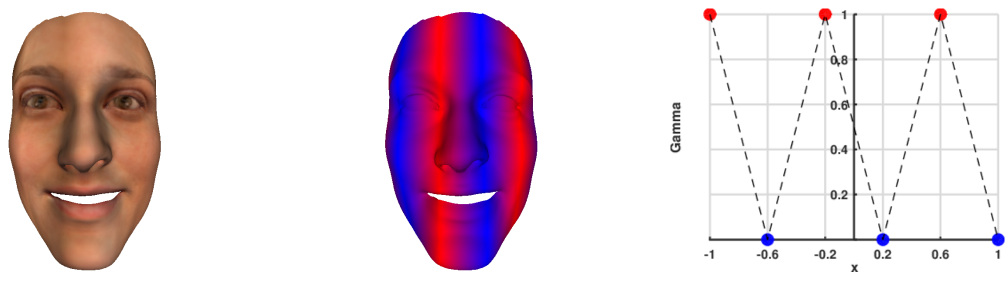


(c) $\gamma = 1.0/0.5/0.0/0.0/0.0$

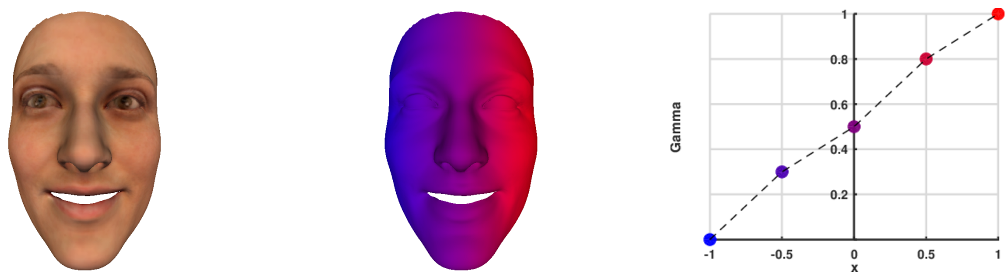
Figure 4.10: Segmented average merge

The examples above show how with segmented average method there is a significant increase on spatial control when merging, without compromising the weighting. The two last scenarios split the x axis into 4 bins, having one side completely dominated by the corresponding side model and the merging happens localized in the other half. This generalized method opens new opportunities as a linear sectioned merge between two arbitrary models. Figure 4.11 shows two exceptional cases, one with interlaced average regions and the another with inverted sides, this last one also possible with a linear merge.

Although this deviates from the purpose of our left-right applications and results might not be as desired, demonstrates the powerful flexibility this tool could have for future use cases.



(a) $\gamma = 1.0/1.0/1.0/0.5/0.0$



(b) $\gamma = 1.0/1.0/0.5/0.3/0.0$

Figure 4.11: Segmented generalized merge

Chapter 5

Match and Replace

This chapter develops a technique to replace a given region from one model into another. Figure 5.1 shows the top level sequential flow diagram, all included in the script *replace.py* (see Section 7.5). The algorithm starts from fetching and loading the models, later performs an alignment between models and finally the actual replacement before saving the results into a new file.

Figure 5.2 shows a graphical clarification of the match and replace process, a *region* is defined and replaced on the *target* model with the contents of the *source* model.

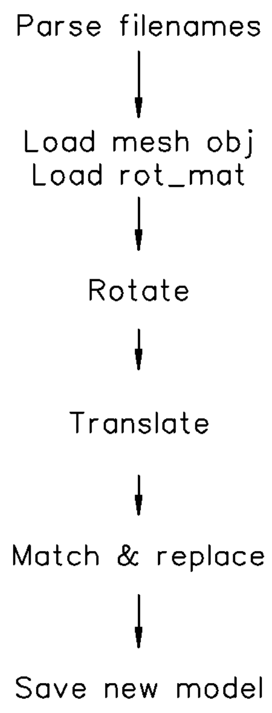


Figure 5.1: Merging sequence

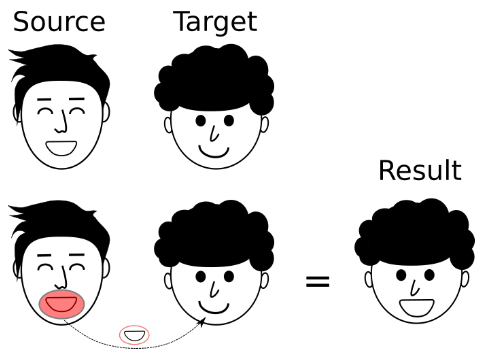


Figure 5.2: Match and replace

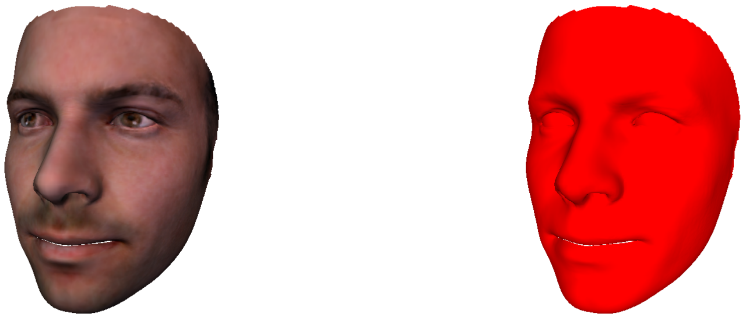
The next sections present details regarding these steps. To illustrate the process, examples of paired models are provided, auxiliary colored models are included as heat maps to reinforce visualization throughout the process.

5.1 Loading the files

The program takes two arguments, the *target* and *source* files without file extension, to load the mesh objects and the rotation matrix files, the filenames are generated internally by adding the corresponding `.obj` and `.mat` suffixes:

Unlike the merging process, the mesh files do not include the "_mesh.obj" from the 3D conversion (see Section 4.1). The replace tool is intended to be generic and used after any merging (as depicted in the scope, see Figure 1.1), the *merge* program removes and changes this suffix on the output file.

Note also that the two models belonging to this process are not from the same person, though are intended to be similar. The purpose of the match and replace is to identify a damaged region on the target model and replace it with the same region from the source. Figure 5.3 shows the model for two different persons, as convenience we'll use the *red* model as the target and the *blue* as the source on the heat maps.


(a) Target model

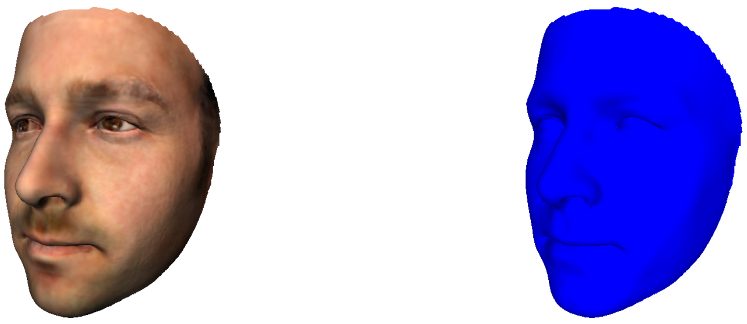

(b) Source model

Figure 5.3: Match and replace models

5.2 Alignment

If the input models for this *replace* stage are generated after a *merge* process, the two models are likely to be aligned. However, the script does align the two models before the match and replace, since is intended to be a generic tool and to avoid dependency from the merge script. The models follow identical alignment steps as described in Section 4.2, and thus the rotation matrix files are also required to be located along with the mesh files.

5.3 Averaging

In addition, Section 4.4 uses a γ factor to weight how one model merges with the another, its behavior is further developed in the subsequent sections. This method also uses such factor when doing the replacement, being γ the weight of the **target** model into the result. Its range is also [0,1], meaning when $\gamma = 1$ the model trends to the target, while a $\gamma = 0$ value makes the source dominant. The script *replace.py* accepts defining a *gamma* value, or array of values, as demonstrated in Section 4.6, this is for example

```
python replace.py target source --gamma 0.0/0.5/1.0
```

Note the order the γ values are declared. Recalling the *merge* model, these values represent points from left to right ($x = -1 \rightarrow x = 1$), here during replacement, these span from the center of the region outwards ($r' = 0 \rightarrow r' = r$). Therefore, the initial γ value 0.0 indicates the region is source-dominant.

5.4 Region matching

A *region* is the intersection between two volumes, the first one being the face model and the second an arbitrary selection object. Notice the last object is not the source model, we'll use two regular shapes: a sphere and a cuboid, their intersections trend to be circles, ellipses, rectangles and parallelograms.[1] Characterize these selection volumes requires three variables, namely

1. *Mode*: Sets the area shape to be spherical (default) or rectangular.

2. *Center*: Corresponds to the $< x, y, z >$ coordinates of the center of the region. The default value is < 0.0,0.0,0.25 >, and if a dimension is omitted, a value of zero is used instead.

3. *Radius*: The extent of the area. When `--mode rect` is called it can be a three-value parameter indicating each $< x, y, z >$ axis radius, if a dimension is omitted, the value is cloned from the other dimensions. When `--mode sphere` is used only one value is required. The default value is 0.1.

The script *replace.py* accepts passing tuple values separated by slash ("/"), for example the region defined as

```
python replace.py tg src --center 0.35/0.25/0.15 --radius 0.25 --gamma 0.0
```

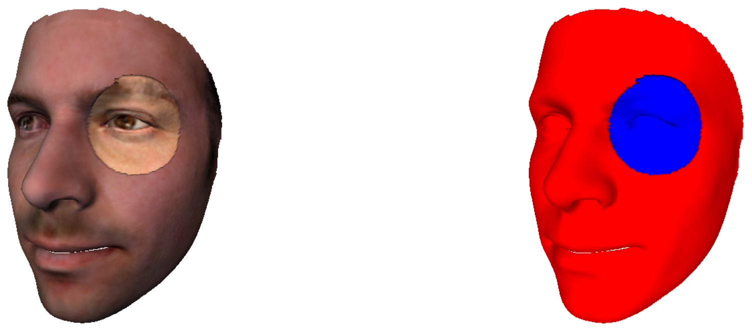
produces the output shown in Figure 5.4a, where a $\gamma = 0.0$ value indicates no average is performed, and the region is filled entirely with the source model. The equivalent replacement using `rect` mode is shown in 5.4b.

Comments on the region area:

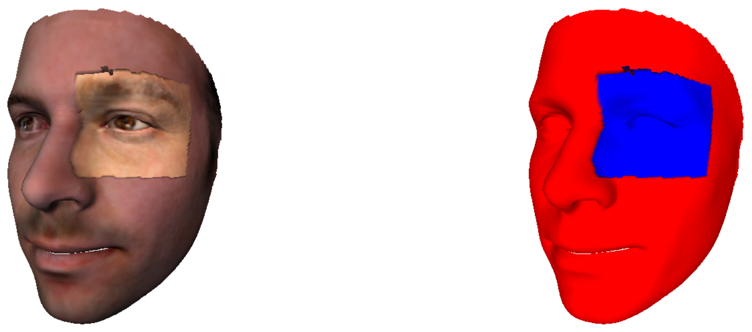
- At the same `radius` value, the `rect` mode results on a bigger area than the `sphere` mode, since the latter produces a region that would be inscribed into the former.

- The `center` point can land outside the face model shell, together with the `radius` parameters define the area extension. Any point inside this area is processed by the replacement algorithm.

[1]Note the irregular shape of the face influences greatly on the intersected region, could also create non-contiguous regions. In this project are used typical scenarios that may not reflect this effect.



(a) Spherical



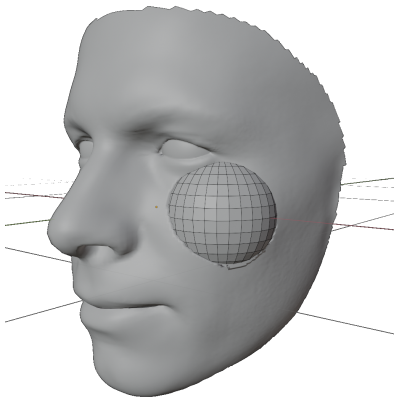
(b) Rectangular

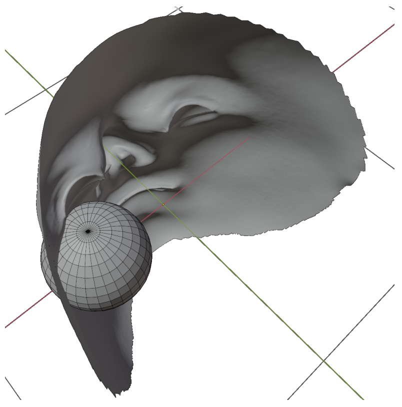
Figure 5.4: Region mode

- The two modes are provided to ease the selection of a given face element. For example mouth and forehead portions are better matched using rectangular shapes, eyes and cheeks could go better with an spherical shape, while the nose works fine with both. The mode is intended to increase flexibility to the matching step, while the spherical produces better results when averaging, as discussed in the next sections.

5.5 Spherical mode

The replacement region is computed as the intersection between the target model and a sphere of radius r, which is centered at an arbitrary point $< x, y, z >$ in world coordinates, as shown in Figure 5.5. For vertices inside the sphere volume ($r' < r$) their properties are replaced and averaged with the source model, vertices outside the sphere ($r' > r$) are not considered. The parameter $\gamma = f(r)$ indicates the weight of the target model into the resulting object as function of the radius. When several γ values are specified, the initial value corresponds to $r' = 0$, whereas the last one at $r' = r$. Linear and segmented average methods provide a smoother transitions at the edge. Figure 5.6 includes examples of spherical regions with different center, radius and gamma values, the auxiliary polar plots indicate the gamma value as function of the radius.


(a) Front view


(b) Rear view

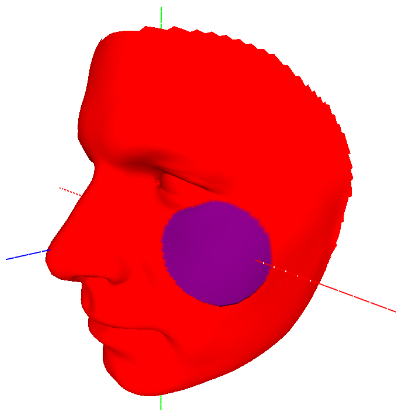
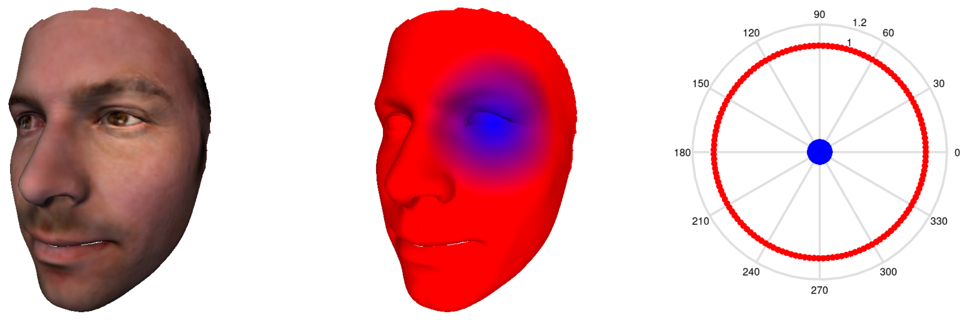
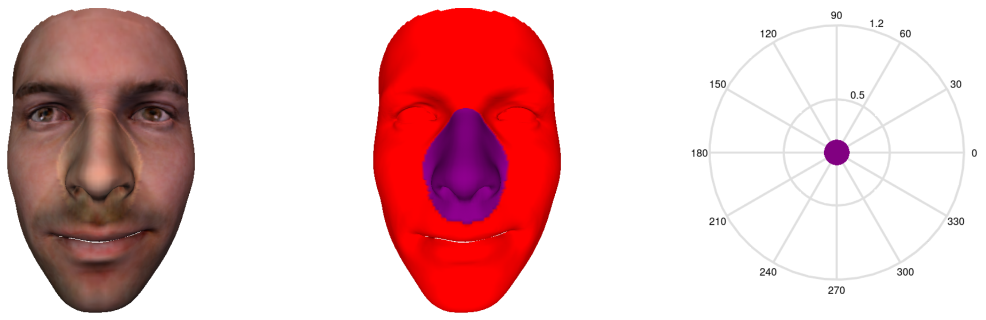

(c) Match
$c = 0.4/0.0/0.0, r = 0.25$

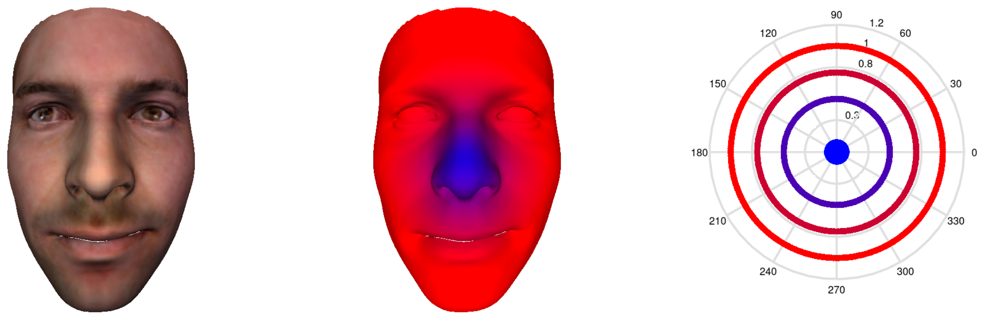
Figure 5.5: Spherical intersection



(a) $c = 0.35/0.25/0.15, r = 0.4, \gamma = 0.0/1.0$



(b) $c = 0.0/0.0/0.45, r = 0.3, \gamma = 0.5$

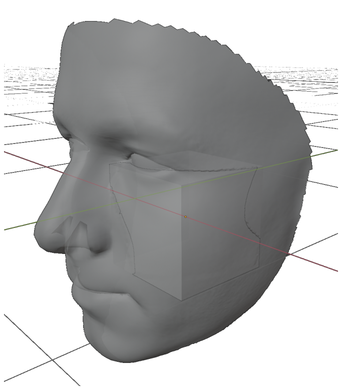


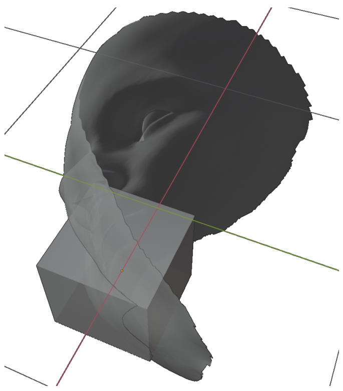
(c) $c = 0.0/0.0/0.50, r = 0.6, \gamma = 0.0/0.3/0.8/1.0$

Figure 5.6: Sphere mode examples

5.6 Rectangular mode

The replacement region is computed as the intersection between the target model and a cuboid with radius $r =< r_x, r_y, r_z >$ on all dimensions, which is centered at an arbitrary point $p =< x, y, z >$ in world coordinates, as shown in Figure 5.7. For vertices inside the rectangular volume $(x < r_x \,\&\, y < r_y \,\&\, z < r_z)$ their properties are replaced and averaged with the source model, vertices outside the box $(x > r_x \,|\, y > r_y \,|\, z > r_z)$ are not considered. The parameter $\gamma = f(r)$ indicates the weight of the target model into the resulting object as function of the radius. When several γ values are specified, the initial value corresponds to $r' = 0$, whereas the last one at $r' = r$. Linear and segmented average methods provide a smoother transitions at the edge. Figure 5.8 includes examples of rectangular regions with different center, radius and gamma values, the auxiliary plots indicate the gamma values as function of the radius.


(a) Front view


(b) Rear view

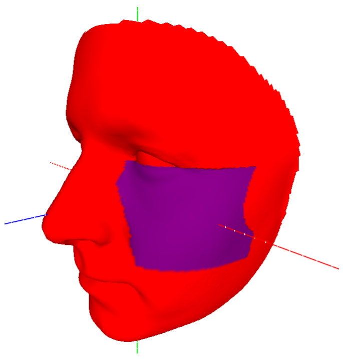
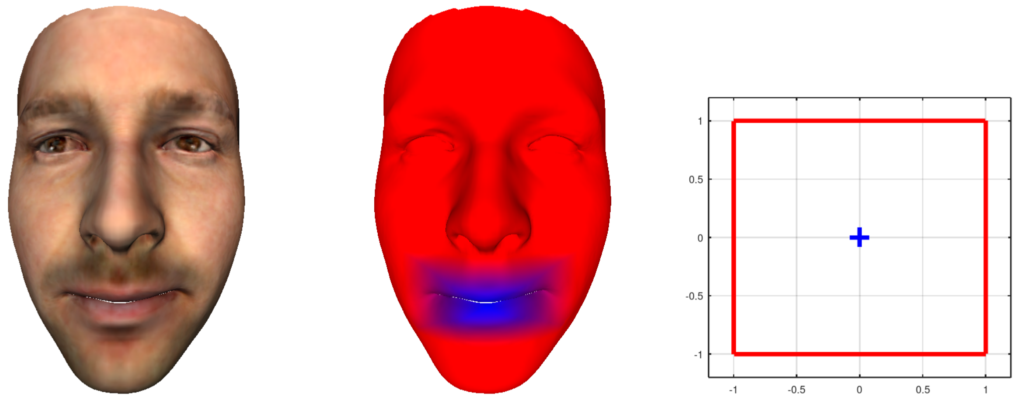
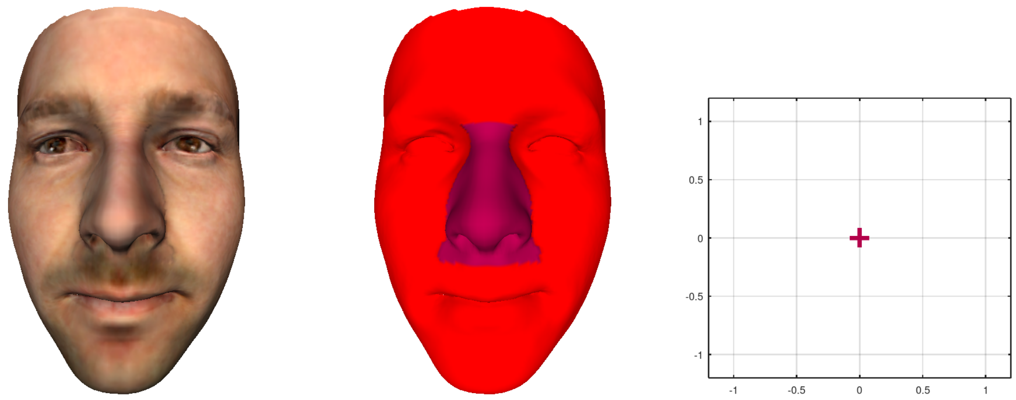

(c) Match
$c = 0.4/0.0/0.0, r = 0.25$

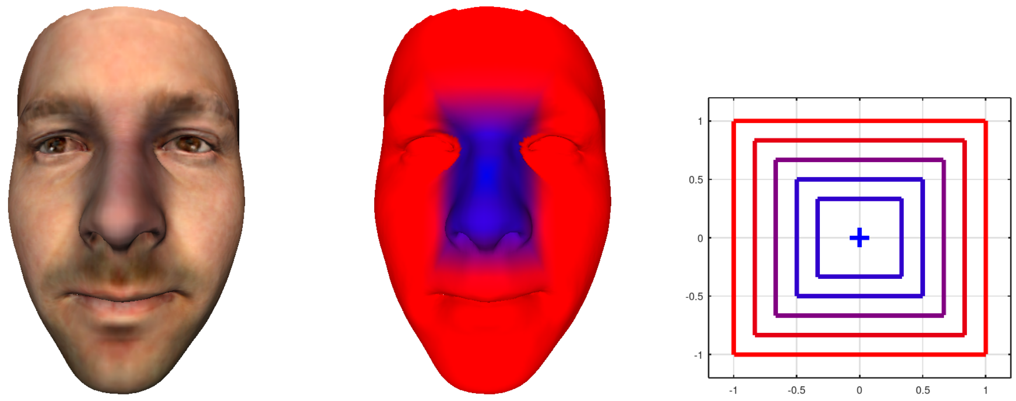
Figure 5.7: Rectangular intersection



(a) $c = 0/-0.42/0.28, r = 0.4/0.2/0.15, \gamma = 0.0/1.0$



(b) $c = 0.0/0.05/0.475, r = 0.22/0.3/0.2, \gamma = 0.7$



(c) $c = 0.0/0.1/0.5, r = 0.3/0.5/0.3, \gamma = 0.0/0.2/0.2/0.5/0.9/1.0$

Figure 5.8: Rectangular mode examples

Chapter 6

Conclusions

This section highlights the outcomes of this project as well as final thoughts and other observations. The scope of this project (Figure 1.1) involved several blocks between the path from paired photos of a person to unified conditioned three-dimensional model. Below is a summary of findings regarding these components and their characteristics.

- The modern Face reconstruction modeler utilized takes a single 2D photo as input files and generate a full-blown 3D model of the face. The reconstructor implements a neural network attached to a standard 3D model that adjusts its shape based upon the information extracted from the 2D picture. The algorithm uses face landmark information to form a pose and expression basis that together with a shape modeler generate an accurate three-dimensional representation of the face.

- The output model is a Mesh object that includes position and color texture information. A quick overview on graphical elements describes the type of elements that constitute the face object, how this information is saved into a wavefront `.obj` file, and presents a simple python class to handle the object and perform basic operations.

- As two independent models (left and right) are generated separately, a Merge tool is developed to unify both models into a single one. To achieve this, is required to know the rotation matrix parameter of each object, in order to perform correctly the alignment between the two, and place them in the same coordinate system. Moreover, a constant merge when combining produces an enhanced version of the coarse merge, where no abrupt edges are visible and the information of the two sides is considered. To have more spatial control, a linear average merge technique is proposed to intensify details belonging to each side, while having a smooth transition at the midpoint. The segmented linear merge demonstrates a generalization of this behavior by providing a flexible scheme for more complex outcomes.

- Another implemented method performs Segmented linear merge onto a desired area with information from another model. This second technique uses a Region matching process to select the area to be replaced and perform also Averaging between the models similar to the merging tool. Two selection modes are provided to fit different face areas. The Spherical mode provides homogeneous averaging with smoother transitions in a radial manner, while the Rectangular mode is suited for more delimited regions such as mouth and forehead.

6.1 Observations

There are some considerations and limitations identified along the course,

- The face reconstructor does not include ears, hair or neck information into the model, as denoted in [2], nor have details on the teeth. In addition, it was demonstrated that a seriously damaged face does not reflect into the 3D model, as expected since the recontructor does not handle any information of this kind, though the face is greatly deformed due to the injury.
- The code can generate the heat maps used in the examples on this report, by passing the `--heat` fuse as argument to the *merge.py* and *replace.py* programs, see Section 7.
- The auxiliary Gamma-γ Maps on the examples are generated using GNU Octave and the source files are included in Section 7.7.

6.2 Future work

As discussed in Section 1, this project is within a much larger project framework and thus, many aspects can be improved and attached to the pipeline developed on it. For example from expand the deep learning modules to consider and handle missing face components, as commented in the previous section, to improve the efficiency and error handling in the written code.

One closer improvement related to this work is the generation of the selection area during *matching* and the specification of the γ points. For this, a graphical software that superposes the models and provides the required information, such as *center* and *radius* values, is undoubtedly a powerful tool for accelerating the curve when setting this parameters, considering also non-technical users.

Chapter 7

Code

The following files were developed as part of this project.

7.1 Header

```
#!/usr/bin/env python3
#-*- coding: utf-8 -*-

#MIT License

#Copyright (c) 2022 Jose Villalobos

#Permission is hereby granted, free of charge, to any person
#obtaining a copy of this software and associated documentation
#files (the "Software"), to deal in the Software without restriction,
#including without limitation the rights to use, copy, modify, merge,
#publish, distribute, sublicense, and/or sell copies of the Software,
#and to permit persons to whom the Software is furnished to do so,
#subject to the following conditions:

#The above copyright notice and this permission notice shall be
#included in all copies or substantial portions of the Software.

#THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
#EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
#OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
#NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
#BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
#ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
#CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
#SOFTWARE.
```

7.2 Image preprocessing

code/preproc.py

```python
import cv2
import os
import glob
import io
from mtcnn import MTCNN

detector = MTCNN()
color = (170,0,255)
image_path = 'input'
save_path = 'preproc'

if not os.path.exists(save_path):
    os.makedirs(save_path)

img_list = glob.glob(image_path + '/' + '*.jpg')

for img in img_list:

    print("Pre-processing %s" % img)
    image = cv2.cvtColor(cv2.imread(img), cv2.COLOR_BGR2RGB)
    result = detector.detect_faces(image)
    bounding_box = result[0]['box']
    keypoints = result[0]['keypoints']

    cv2.rectangle(image,
                  (bounding_box[0], bounding_box[1]),
                  (bounding_box[0]+bounding_box[2], \
                      bounding_box[1] + bounding_box[3]),
                  color,
                  8)

    cv2.circle(image,(keypoints['left_eye']), 8, color, 8)
    cv2.circle(image,(keypoints['right_eye']), 8, color, 8)
    cv2.circle(image,(keypoints['nose']), 8, color, 8)
    cv2.circle(image,(keypoints['mouth_left']), 8, color, 8)
    cv2.circle(image,(keypoints['mouth_right']), 8, color, 8)

    cv2.imwrite(save_path+"/"+os.path.basename(os.path.splitext(img)[0]) + \
        "_mtcnn.jpg", cv2.cvtColor(image, cv2.COLOR_RGB2BGR))
    f = open(save_path + "/" + os.path.basename( \
            os.path.splitext(img)[0]) + ".txt","w")

    # initializing delim
    delim = "    "

    f.write(delim.join(map(str, keypoints['left_eye']))+"\n"+
            delim.join(map(str, keypoints['right_eye']))+"\n"+
            delim.join(map(str, keypoints['nose']))+"\n"+
            delim.join(map(str, keypoints['mouth_left']))+"\n"+
            delim.join(map(str, keypoints['mouth_right'])))
    f.close()

    print(result)
```

7.3 OBJ class

code/obj.py

```
import io

class obj:
    def __init__(self, label=None):
        self.label = label
        self.vertices = []
        self.faces = []

    def load(self, path):
        with open(path,'r') as file:
            if (file):
                for line in file:
                    l = line.strip('\n').split(' ')
                    if line[0] == 'v':
                        self.vertices.append( [float(i) for i in l[1:]] )
                    if line[0] == 'f':
                        self.faces.append( [ int(i) for i in l[1:]] )
            else :
                print("Can't read file",end="\n")


    def save(self, path):
        with open(path,'w+') as file:
            for vert in self.vertices:
                file.write('v %f %f %f %f %f %f\n' % \
                    (vert[0],vert[1],vert[2],vert[3],vert[4],vert[5]))
            file.write('\n')
            for f in self.faces:
                file.write('f %i %i %i\n'%(f[0],f[1],f[2]))
            file.close()

    def limits(self):
        seed = self.vertices[0][0:3]
        # v saves the three coordinates for each coordinate evaluation
        vmin = [seed, seed, seed]
        vmax = [seed, seed, seed]
        for vertex in self.vertices[1:]:
            # X coordinate
            if vertex[0] < vmin[0][0]: vmin[0] = vertex[0:3]
            if vertex[0] > vmax[0][0]: vmax[0] = vertex[0:3]
            # Y coordinate
            if vertex[1] < vmin[1][1]: vmin[1] = vertex[0:3]
            if vertex[1] > vmax[1][1]: vmax[1] = vertex[0:3]
            # Z coordinate
            if vertex[2] < vmin[2][2]: vmin[2] = vertex[0:3]
            if vertex[2] > vmax[2][2]: vmax[2] = vertex[0:3]
        return vmin, vmax


    def rotate(self, matrix):
        for vertex in self.vertices:
            x = vertex[0]*matrix[0][0] + \
                vertex[1]*matrix[0][1] + \
                vertex[2]*matrix[0][2]


            y = vertex[0]*matrix[1][0] + \
                vertex[1]*matrix[1][1] + \
                vertex[2]*matrix[1][2]

            z = vertex[0]*matrix[2][0] + \
                vertex[1]*matrix[2][1] + \
                vertex[2]*matrix[2][2]

            vertex[0] = x
            vertex[1] = y
            vertex[2] = z


    def translate(self, x, y, z):
        for vertex in self.vertices:
            vertex[0] = vertex[0] + x
            vertex[1] = vertex[1] + y
            vertex[2] = vertex[2] + z
```

7.4 Merge

```
usage: merge.py [-h] [--gamma GAMMA] [--heat] bfn

Merge left and right face meshes

positional arguments:
  bfn            Base filename

optional arguments:
  -h, --help     show this help message and exit
  --gamma GAMMA  Merging factor
  --heat         Turn on heat color
```

code/merge.py

```python
import io
import os
import argparse
import math
import scipy.io
import numpy as np
from obj import obj

def parse_args():

    desc = "Merge left and right face meshes"
    parser = argparse.ArgumentParser(description=desc)

    parser.add_argument('bfn', type=str, default=None, help='Base filename')
    parser.add_argument('--gamma', type=str, default='0.5', help='Merging factor')
    parser.add_argument('--heat', action='store_true', help='Turn on heat color')

    return parser.parse_args()



def getGamma(gamma, px, xmin, xmax):
    l = len(gamma) - 1
    if(l == 0): return gamma[0]
    else:
        total = (xmax - xmin)
        bin_size = total / l
        bin_n = (px - xmin) / bin_size
        n = int(math.floor(bin_n))
        if (n == l): return gamma[-1]
        else:
            a = gamma[n]
            b = gamma[n+1]
            return (b-a)*(bin_n-n)+a






```

```
def merge():

    args = parse_args()

    if (args.heat):
        import utils

    # Compute gamma
    g = str(args.gamma).split("/")
    gamma = list(map(lambda x: abs(float(x)),g))
    print("Gamma: ", gamma)

    # Parse the filenames required files
    if not args.bfn:
        print("Filename basename is missing")
        return 0
    else:
        left, right = obj("left"), obj("right")
        for m in [left,right]:
            fn_mesh = args.bfn+"-"+m.label+"_mesh.obj"
            fn_mat = args.bfn+"-"+m.label+".mat"
            print("Obj: %s" % fn_mesh, end=('\n'))
            print("Mat: %s" % fn_mat, end=('\n'))

            # Create and load the mesh
            m.load(fn_mesh)

            # Load the rotation matrix
            f_mat = scipy.io.loadmat(fn_mat)
            rot = f_mat['rot_mat']

            # If heat map on, change color
            if (args.heat):
                utils.changeColor(m, utils.heat_colors[m.label])

            # Rotate
            m.rotate(rot)

            # Found the box and translate center
            vmin, vmax = m.limits()
            off_x = (vmax[0][0] + vmin[0][0]) / 2
            off_y = (vmax[1][1] + vmin[1][1]) / 2
            off_z = (vmax[2][2] + vmin[2][2]) / 2

            m.translate(-off_x, -off_y, -off_z)

            # Save the limits in x
            m.xmin = vmin[0][0] - off_x
            m.xmax = vmax[0][0] - off_x

        # Set up merge
        left_len = len(left.vertices)
        right_len = len(right.vertices)
        if (left_len != right_len): print("Warning: Mesh size unmatch")

        merged = obj()
        merged.faces = left.faces # Inherit faces
        # Merge
        for i in range(0,min(left_len,right_len)):
            v = []
            g = getGamma(gamma, left.vertices[i][0], left.xmin, left.xmax)
            for j in range(0,6):
                v.append( left.vertices[i][j] * g  + right.vertices[i][j] * (1-g)
    )
            merged.vertices.append(v)

        # Save results
        merged.save(args.bfn+"-merged.obj")


if __name__ == '__main__':
    merge()
```

7.5 Match and Replace

```
usage: replace.py [-h] [--center CENTER] [--radius RADIUS] [--gamma GAMMA]
                  [--heat] [--mode MODE]
                  target source

Merge left and right face meshes

positional arguments:
  target           Target filename w/o extension
  source           Source filename w/o extension

optional arguments:
  -h, --help       show this help message and exit
  --center CENTER  Replacement center point
  --radius RADIUS  Replacement radius
  --gamma GAMMA    Merging factor
  --heat           Turn on heat color
  --mode MODE      Turn on heat color
```

code/replace.py

```python
import io
import os
import argparse
import math
import scipy.io
import numpy as np
from obj import obj


def parse_args():

    desc = "Merge left and right face meshes"
    parser = argparse.ArgumentParser(description=desc)

    parser.add_argument('target', type=str, default=None, \
        help='Target filename w/o extension')
    parser.add_argument('source', type=str, default=None, \
        help='Source filename w/o extension')
    parser.add_argument('--center', type=str, default='0/0/0.25', \
        help='Replacement center point')
    parser.add_argument('--radius', type=str, default='0.1', \
        help='Replacement radius')
    parser.add_argument('--gamma', type=str, default='0.5', \
        help='Merging factor')
    parser.add_argument('--heat', action='store_true', \
        help='Turn on heat color')
    parser.add_argument('--mode', type=str, default='sphere', \
        help='Turn on heat color')

    return parser.parse_args()


```

```python
# p = point <x,y,z>, c = center <x,y,z>, r = radius <x,y,z>
def getGamma(gamma, p, c, r, mode='sphere'):
    l = len(gamma) - 1
    d = [0] * 3
    d[0] = abs(p[0] - c[0])
    d[1] = abs(p[1] - c[1])
    d[2] = abs(p[2] - c[2])
    dd = d[0]*d[0]+d[1]*d[1]+d[2]*d[2]

    if (mode == 'sphere') :
        if ( dd < r[3] ):
            if (l == 0) : return gamma[0]
            else:
                bin_size = math.sqrt(r[3]) / l
                bin_n = math.sqrt(dd) / bin_size
                n = int(math.floor(bin_n))
                if (n >= l): return gamma[-1]
                else:
                    a = gamma[n]
                    b = gamma[n+1]
                    return (b-a) * (bin_n-n) + a

    if (mode == 'rect') :
        if ( d[0] < r[0] and d[1] < r[1] and d[2] < r[2] ):
            if (l == 0) : return gamma[0]
            else:
                g = []
                n = 0
                bn = 0
                for axis in range(0,2):
                    total = r[axis]
                    bin_size = total / l
                    bin_n = d[axis] / bin_size
                    ni = int(math.floor(bin_n))
                    if (ni > n): n = ni
                    if (bin_n > bn): bn = bin_n
                if (n >= l): return gamma[-1]
                else:
                    a = gamma[n]
                    b = gamma[n+1]
                    return (b-a)*(bn-n)+a

    return 1


def replace():

    args = parse_args()

    if (args.heat):
        import utils

    # Compute gamma
    g = str(args.gamma).split("/")
    gamma = list(map(lambda x: abs(float(x)),g))
    print("Gamma: ", gamma)

    # Parse the filenames required files
    if not args.source or not args.target:
        print("Filename missing")
        return 0
    else:
        source, target = obj("source"), obj("target")
        for m in [source,target]:
            fn_mesh = getattr(args,m.label)+".obj"
            fn_mat = getattr(args,m.label)+".mat"
            print("Obj: %s" % fn_mesh, end=('\n'))
            print("Mat: %s" % fn_mat, end=('\n'))

            # Create and load the mesh
            m.load(fn_mesh)

            # Load the rotation matrix
            f_mat = scipy.io.loadmat(fn_mat)
            rot = f_mat['rot_mat']

            # If heat map on, change color
            if (args.heat):
                utils.changeColor(m, utils.heat_colors[m.label])

            # Rotate
            m.rotate(rot)

            # Find the box and translate center
            vmin, vmax = m.limits()
            off_x = (vmax[0][0] + vmin[0][0]) / 2
            off_y = (vmax[1][1] + vmin[1][1]) / 2
            off_z = (vmax[2][2] + vmin[2][2]) / 2

            m.translate(-off_x, -off_y, -off_z)

        # Set up replace
        src_len = len(source.vertices)
        target_len = len(target.vertices)
        if (src_len != target_len): print("Warning: Mesh size unmatch")

        # Compute center point (magnet to target)
        center = str(args.center).split("/")
        center = list(map(lambda x: float(x), center))

        # Fill empty parameters
        if (len(center) == 1): center.append(0)
        if (len(center) == 2): center.append(0)
        print("Center: ", center)

        # Compute radius
        radius = str(args.radius).split("/")
        radius = list(map(lambda x: abs(float(x)), radius))

        # Fill empty parameters
        if (len(radius) == 1): radius.append(radius[0])
        if (len(radius) == 2): radius.append(radius[-1])
        radius.append(radius[0]*radius[0])
        print("Radius: ", radius)

        # Set up replace
        print("Mode: ",args.mode)
        replaced = obj()
        replaced.faces = target.faces # Inherit faces



        # Replace
        for i in range(0,min(src_len,target_len)):
            v = []
            g = getGamma(gamma, target.vertices[i], center, radius, args.mode)
            for j in range(0,6):
                v.append( \
                    target.vertices[i][j] * g  + \
                    source.vertices[i][j] * (1-g) )
            replaced.vertices.append(v)

        # Save results
        replaced.save(args.target+"-replaced.obj")


if __name__ == '__main__':
    replace()
```

7.6 Heat Map utilities

code/utils.py

```python
heat_colors = { "left":[1,0,0],    \
                "right":[0,0,1],   \
                "target":[1,0,0],  \
                "source":[0,0,1]
              }

def changeColor(obj, RGB):
    for v in obj.vertices:
        v[3] = RGB[0]
        v[4] = RGB[1]
        v[5] = RGB[2]
```

7.7 Gamma Map utilities

code/segmented.m

```
clc;
clear;

g = [1.0,0.0,1.0,0.0,1.0,0.0];
x = linspace(-1,1,length(g))

figure(1)
gt = transpose(g);
hold on;
scatter(x,g, 200, [gt,0*gt,1-gt], "filled")
plot(x,g,"color","black", "linestyle", "--","linewidth", 1)
hold off;
set (gca, "ylim", [0 1])
set (gca, "xtick", x)
set (gca, "xgrid", "on")
set (gca, "ygrid", "on")
set (gca, "fontweight", "bold")
set (gca, "xlabel", "x")
set (gca, "ylabel", "Gamma")
set (gca, "yaxislocation", "origin")
set (gca, "linewidth", 2)
set (gca, "fontsize", 14)
```

code/polarmaps.m

```
 clf;
 clc;
 clear;

 gamma = [0.0,1.0]
% gamma = [0.5];
% gamma = [0.0,0.3,0.8,1.0]
% gamma = [1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0]
 l = length(gamma)
 for i = 1:l
   gt = gamma(i)
   if (i == 1)
     n = 1;
     theta = 0;
     h = polar (theta, 0, ".");
     set (h, "markersize", 100);
   else
     n = 72*i;
     r = i/l
     theta = linspace (0,2*pi,n);
     h = polar (theta, r*ones(n), ".");
     set (h, "markersize", 25);
 endif
   hold on;
   set (h, "linewidth", 2);
   set (h, "color", [gt,0,1-gt]);

 endfor;
set (gca, "rtick", [gamma,1.2])
set (gca, "linewidth", 3)
set (gca, "fontsize", 18)
```

code/rectmaps.m

```
clf;
clc;
clear;

 gamma = [0.0,1.0]
% gamma = [0.7]
% gamma = [0.0,0.2,0.2,0.5,0.9,1.0];
% gamma = [1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0]
l = length(gamma)
for i = 1:l
  gt = gamma(i)
  if (i == 1)
    h = plot (0, 0, "+");
    set (h, "markersize", 25);
  else
    r = i/l
    x = [-r,r, r,-r, -r];
    y = [ r,r,-r,-r,  r];
    h = plot(x,y,"-");
  endif
  hold on;
  set (h, "linewidth", 6);
  set (h, "color", [gt,0,1-gt]);

endfor;
set (gca, "xlim", [-1.2,1.2])
set (gca, "ylim", [-1.2,1.2])
set (gca, "linewidth", 2)
set (gca, "fontsize", 18)
set (gca, "xgrid", "on")
set (gca, "ygrid", "on")
```

Appendix A

Software versions

Below are tracked the software versions used during the development of this project.

Debian GNU/Linux

```
Linux debian 5.10.0-11-amd64 #1 SMP Debian 5.10.92-1 (2022-01-18) x86_64
Distributor ID: Debian
Description:    Debian GNU/Linux 11 (bullseye)
Release:        11
Codename:       bullseye
```

Python

```
Python 3.6.13 :: Anaconda, Inc.
```

Anaconda

```
conda 4.9.2
```

TensorFlow

```
Version: 1.12.0
```

Numpy

```
Version: 1.19.5
```

Scipy

```
Version: 1.5.4
```

MeshLab

```
MeshLab_64bit_fp v2020.12
built on Dec 1 2020 with GCC 5.5.0 and Qt 5.12.9
```

Appendix B

Base directory content tree

Directory content structure after following the instructions on [2]. Only three levels deep are shown. Note some .tar files might be kept after extraction and some .pdf files are additionally downloaded from attached links on the repository.

```
google/
|-- tf_mesh_renderer
|-- CONTRIBUTING.md
|-- LICENSE
|-- README.md
|-- WORKSPACE
|-- mesh_renderer
|-- runtests.sh
|-- third_party
```

```
juyong/
|-- 1708.00980.pdf
|-- 3DFace
|   |-- LICENSE
|   |-- README.md
|-- Coarse_Dataset
|   |-- CoarseData.zip
|   |-- Exp_Pca.bin
|   |-- ReadMe
|   |-- Sample_code.cpp
|-- Coarse_Dataset.zip
|-- Fine_Dataset.zip
```

```
microsoft/
|-- Deep3DFaceReconstruction
|   |-- BFM
|   |-- LICENSE
|   |-- data_loader.py
|   |-- demo.py
|   |-- face_decoder.py
|   |-- images
|   |-- inception_resnet_v1.py
|   |-- input
|   |-- losses.py
|   |-- network
|   |-- networks.py
|   |-- options.py
|   |-- output
|   |-- preprocess_img.py
|   |-- processed_data
|   |-- readme.md
|   |-- reconstruction_model.py
|   |-- renderer
|   |-- skin.py
|   |-- train.py
|   |-- utils.py
|   |-- weights
|-- FaceReconModel
|   |-- FaceReconModel.data-00000-of-00001
|   |-- FaceReconModel.index
|   |-- FaceReconModel.meta
|   |-- FaceReconModel.pb
|-- FaceReconModel.zip
|-- rasterize_triangles_kernel.so
```

```
mtcnn/
|-- AUTHORS
|-- LICENSE
|-- MANIFEST.in
|-- README.rst
|-- example.ipynb
|-- example.py
|-- ivan.jpg
|-- ivan_drawn.jpg
|-- mtcnn
|   |-- __init__.py
|   |-- data
|   |-- exceptions
|   |-- layer_factory.py
|   |-- mtcnn.py
|   |-- network
|   |-- network.py
|-- no-faces.jpg
|-- requirements.txt
|-- result.jpg
|-- setup.py
|-- tests
    |-- __init__.py
    |-- test_mtcnn.py
```

```
unibas/
|-- BFModel09.pdf
|-- model2019
|   |-- model2019_bfm.h5
|   |-- model2019_face12.h5
|   |-- model2019_fullHead.h5
|   |-- model2019_fullHead_lvl1.ply
|   |-- model2019_fullHead_lvl2.ply
|   |-- model2019_mouthOnly.ply
|   |-- model2019_textureMapping.json
```

Bibliography

[1] Andrea Scotta. «Ricostruzione facciale 3D basata su immagine e il suo possibile utilizo in riabilitazione prostesica». MA thesis. Torino: Universita' degli Studi di Torino, 2020 (cit. on p. 3).

[2] Yu Deng, Jiaolong Yang, Sicheng Xu, Dong Chen, Yunde Jia, and Xin Tong. «Accurate 3D Face Reconstruction with Weakly-Supervised Learning: From Single Image to Image Set». In: *IEEE Computer Vision and Pattern Recognition Workshops*. `https://github.com/Microsoft/Deep3DFaceReconstruction`. 2019 (cit. on pp. 4, 6, 8, 10, 43, 60).

[3] Iván de Paz Centeno <ipazc@unileon.es>. *MTCNN*. `https://github.com/ipazc/mtcnn`. 2019 (cit. on p. 6).

[4] David Sandberg. *Face Recognition using Tensorflow*. `https://github.com/davidsandberg/facenet`. 2016 (cit. on p. 6).

[5] Kaipeng Zhang, Zhanpeng Zhang, Zhifeng Li, and Yu Qiao. «Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks». In: *IEEE Signal Processing Letters* 23.10 (2016), pp. 1499–1503. DOI: `10.1109/LSP.2016.2603342` (cit. on p. 6).

[6] P. Pascal, R. Knothe, B. Amberg, S. Romdhani, and T. Vetter. *A 3D Face Model for Pose and Illumination Invariant Face Recognition*. Tech. rep. Universität Basel, 2009 (cit. on p. 8).

[7] GNU Octave. *Simple File I/O*. URL: `https://octave.org/doc/v6.4.0/Simple-File-I_002fO.html` (cit. on p. 11).

[8] Scipy. *scipy.io.loadmat — Load MATLAB file*. URL: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.io.loadmat.html` (cit. on p. 11).

[9] Khronos Group. «Primitive». In: *OpenGL Rendering Pipeline*. `https://www.khronos.org/opengl/wiki/Primitive`. 2020 (cit. on p. 12).

[10] Wavefront .obj file. *Wavefront .obj file — Wikipedia, The Free Encyclopedia*. 2020. URL: `https://en.wikipedia.org/wiki/Wavefront_.obj_file` (cit. on p. 18).