University of West Bohemia
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Automatic mesh transformation method for musculoskeletal model

Petr Kellnhofer

Technical Report No. DCSE/TR-2012-06
July, 2012

Distribution: public

Technical Report No. DCSE/TR-2012-06
July 2012

Automatic mesh transformation method for musculoskeletal model

Petr Kellnhofer

Abstract

The roadmap [25] states importance of registration of data sets for creation of the *Virtual Physiological Human*, a model of a human body. It also mentions usage of morphing technique for interpolation of new data. This thesis focuses on the transformations tied with these operations and tries to find an automatic solution which does not need user set up parameters. The deformation filter for surface models of muscles in musculoskeletal model of human body developed in the previous work was chosen as testing application. It has difficulties with damaged input meshes, especially those containing non-manifold edges and vertices. Therefore, the goal is an automatic detection and removal of such artifacts, and the combination of several such inputs into one finer mesh surface gained using a multi-morphing method. To make this possible, approaches for mutual registration of input meshes are analysed, a suitable parametric domain is searched for and appropriate way of final interpolation is chosen. A solution for making such actions in fully automatic manner for general damaged input meshes with similar shape specified by underlying real-world object, but with various initial position and unknown number of topological artifacts consisting of holes and isolated components on top of previously mentioned non-manifold edges and vertices, is then suggested. The resulting method is then implemented and both partial steps and complete design is tested in various experiments. The results are discussed and conclusion is stated at the end of this thesis.

.

Copies of this report are available on
http://www.kiv.zcu.cz/publications/
or by surface mail on request sent to the following address:

University of West Bohemia in Pilsen

Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Contents

List of Figures

Chapter 1

Introduction

Modern medicine collects a lot of data from patients. These data vary from scalar values of a blood pressure, temperature or heart beat through 2D images gained from X-ray to fully 3D images from computer tomography (CT) or magnetic resonance (MR). The data are usually recorded during treatments of individual health issues and are focused on that affected part of the specific patient body. Therefore, we do not have a complete description of the whole body for a single person that would enable us to build a complete model but we rather possess some random samples. It might then be useful to take the missing pieces of data from some general human model adjust them and put them into the model of our patient.

Individual models of some real objects from various data sources can have various positions, rotations and scalings in space. Their combination, however, requires these models to be *registered*. *Registration* is a process that finds relations between a source model and a target model. The result is a transformation function that can transform the source model and this way minimise its difference from the target one. This then enables a projection of properties from one model to another one. Partial information contained on various models can then easily be projected to get one big framework. This can even be applied to data of different types, e.g., 2D image can be registered with 3D surface mesh giving us surface properties missing in the triangular mesh.

Over 150 experts noted this problem in a roadmap [25] leading to creation of *Virtual Physiological Human* (*VPH*) as "a framework of methods and technologies that will make it possible to describe human physiology and pathology in a complete and integrated way." [25]. It requires integration of heterogeneous data, information and knowledge using a global reference system called *Global Reference Body* (*GRB*). It will make it possible to browse, search and analyse all medical data in an easy and unified way. Some experiments are not ethical to

be done on humans. The *GRB* might also enable to project results from similar animal species to the *VPH* model and therefore improve efficiency of medical research [25].

There are more than just space dimensions in a complete data set. Time of data sampling, detail scale, population properties and other dimension descriptors can be adjusted by the viewer of data. This requires suitable user interface for manipulation in such multidimensional space. The population dimension describes space of individual people with the general model in the middle and individuals clustered by common properties like age, weight or blood pressure on the axes [25]. If we could describe these properties and differences between individual models, it would then become possible to extract new models from the existing ones using *morphing* [25]. *Morphing* is a process of interpolation between various models that produces a new model not present in the original input set. In contrast to the registration, the morphing does not consider one model to be the source and the other the target. It takes all objects as equally important points defining the interpolation space so that the new model found somewhere in that space shares some portion of features from all inputs. The portion of similarity to individual inputs can then be adjusted by morphing coefficients. This way, we can, e.g., obtain a model of 60 year old heavy smoker given a model of 50 year and 70 year old patient. If there are more than two input models defining interpolation space then we speak about *multi-morphing*. The *morphing* usually assumes some level of *registration* prior to its running. Therefore those two terms are different but loosely tight to each other.

Creation of *VPH* and *GRB* would enable better human-machine interfaces and modelling of processes in a human body including the pathological ones [25]. That should improve the health care efficiency [25]. The *VPH* can help patients to understand their state, it can help students to learn about human body, it can help doctor to choose proper treatment and it is also supposed to provide a tool for medical research [25]. This is why the *VPH* initiative is regarded so important that it attracted over €200 million of public research funding [25].

In this thesis, I would like to focus on the *registration* and *morphing* problem. The main issue with recent *registration* methods mentioned in [25] is their dependence on user defined parameters. I will therefore look for a fully automatic solution. I am familiar with one specific European Union research project involved in this effort. It is called *VPHOP: Osteoporotic virtual physiological human* (*FP7-ICT-223865*) [26] and it focuses on fight against osteoporosis. In my bachelor thesis [13], I have implemented a *VTK* filter for deformation of surface models of muscles that maintain the volume preservation condition which comes from the incompressible water inside muscles. The method was later improved and published in [17]. Under the above mentioned project, the method was extended with a mutual intersection prevention technique allowing a deformation

of multiple muscles and rigid models of obstacles, formed by bones, at once.

However, the weakness of the method was in the input data quality. It expected closed manifold meshes to be on input of each deformation. Real data was, however, often of a poor quality as a result of errors in input data thresholding and segmentation. The meshes often contained non-manifold edges and details were corrupted in these parts. This led to severe artifacts such those that can be seen in Figure 1.1.

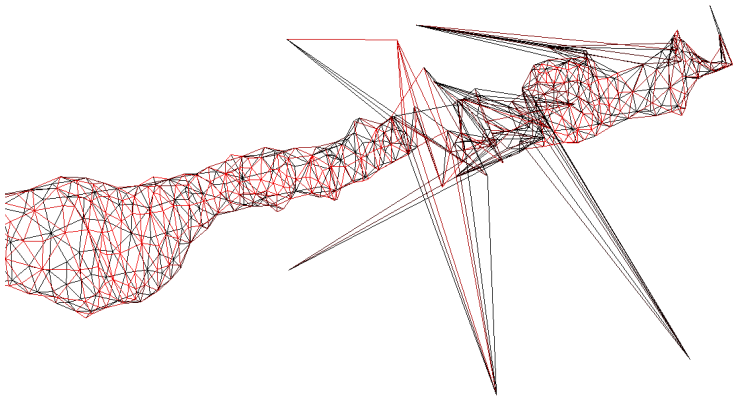


Figure 1.1: Artifacts in product of deformation filter from [13] applied on non-manifold mesh of Sartorius muscle. Taken from [13].

On the other hand, we often have more similar meshes of nearly same object from different sources. I will, therefore, try to find an automatic method that combines multiple surfaces meshes with possibly non-manifold artifacts to create single manifold mesh with better quality. I will try to achieve this with combination of mesh registration techniques for global space registration in general initial pose and multi-mesh morphing for genus 0 closed mesh models. I will have to handle to non-manifold artifact removal which is not done by the above mentioned techniques, as the inputs are usually considered to be closed manifold meshes in the first place. I will focus this technique specifically on the problem of muscle deformation, but the extent of the approach should be broader in reference to the *VPH* initiative.

I will implement created method as a *VTK* filter for a single purpose testing application and experiment with several input sets. I will mainly evaluate the shape preservation quality of output meshes, topological properties and then test their usability in the above mentioned deformation filter. This should lead to an automatic tool usable to processing of data before the application to body modelling tools.

The following two chapters will describe a theoretical background of a surface

mesh registration and multi-morphing. It will also provide an insight into existing methods and evaluate their properties and usability. The Chapter 4 will state a new complex method for the solution of our problem. The Chapter 5 will then present results of experiments and compare them with other methods or parameter settings.

Chapter 2

Mesh registration methods

Sometimes we have more than one model of the same object. Either complete or partial. Often we do not know the position of one object with respect to the others. The registration is then a process that finds a proper transformation for each model or its parts to align models together. For meshes, it usually means to find a location of each vertex on the surface of the other mesh. As the meshes may not have and usually do not have neither the same geometric topology nor number of vertices, vertices usually don't map to vertices. Barycentric coordinates on triangles can help to select the nearest point anywhere on the surface triangle and thus improve the freedom of selection.

There are two main groups of registration methods that differ by the transformation they are trying to find. Rigid methods aim for affine transformation applied to whole mesh. Non-rigid methods have a harder goal when they expect the meshes to be partially or fully deformable and, therefore, rigid transformation has to be found for each vertex.

Following sections further describe examples from both groups and discuss possibility of their application to the problem being solved in this work.

2.1 Rigid methods

Rigid methods assume that both registered meshes have either an identical shape or are partially overlapping subsets of an identical object. This means that purely rigid body transformations are sufficient to align one object to another. In the simplest case, only proper rotation and translation for whole mesh has to be found such this minimises difference between both models after the application of the transformation on all vertices of one of them.

This assumption is usually perfectly valid only for artificial cases of testing meshes

made by cloning one model. In the real situation errors causes that perfect registration cannot be done using rigid transformations. However, for many situations such as registration of 3D scans from different viewpoints, the error may be small enough to be ignored. These methods can then be significantly simpler than those from non-rigid group.

2.1.1 Iterative Closest Point

Iterative Closest Point (*ICP*) is the most common technique for geometric object registration. It was primarily designed for rigid body registration but it became part of various other algorithms, some of them for non-rigid transformations as well.

Basic ICP

The original *Iterative Closest Point* (*ICP*) algorithm was described in [4]. The approach was designed to work with various geometric entities from point sets to parametric surfaces and also for general dimension count. For our needs, the triangle surface mesh representation in 3D space is sufficient.

Input of *ICP* algorithm is then pair of meshes which does not have to have the same number of triangles and vertices, but should have approximately same shape, as the algorithm look for a rigid transformation that moves source mesh P to the best approximation of target mesh X.

Algorithm is based on iterative application of four steps [4]:

1. Matching points of the working mesh P_k to the target mesh X
2. Calculation of a registration function to get the new transformation
3. Application of the resulted transformation to get P_{k+1}
4. Evaluating of the current transformation quality for the algorithm termination

The initial state is given by $P_0 = P$.

In **step 1**, points of the current mesh P_k are assigned best fitting matches from the other mesh X. This fitting is defined by pairs of points with minimal Euclidean distance. Therefore for each point p_i from P_k its image from X is calculated as

$$\overrightarrow{x_i} = \arg\min_{\overrightarrow{x_j} \in X} |\overrightarrow{p_i} - \overrightarrow{x_j}| \tag{2.1}$$

This pairing is then used in **step 2**, where the cross-covariance matrix Σ_{px} is first calculated as

$$\Sigma_{px} = \frac{1}{N_P} \sum_{i=0}^{N_P} \left[(\overrightarrow{p_i} - \overrightarrow{\mu_{Pk}}) \cdot (\overrightarrow{x_i} - \overrightarrow{\mu_X})^T \right] = \frac{1}{N_P} \sum_{i=0}^{N_P} \left[\overrightarrow{p_i} \cdot \overrightarrow{x_i}^T \right] - \overrightarrow{\mu_{Pk}} \cdot \overrightarrow{\mu_X}^T \quad (2.2)$$

where N_P is number of vertices in input mesh $P = P_0$ and therefore all consequent meshes P_k, $x_i \in X$ denotes nearest vertex for $p_i \in P_k$ calculated by 2.1 and $\overrightarrow{\mu_{Pk}}$ with $\overrightarrow{\mu_X}$ are centres of masses of respective meshes given by simple arithmetic average as

$$\overrightarrow{\mu_A} = \frac{1}{|A|} \sum_{\overrightarrow{a} \in A} \overrightarrow{a} \quad (2.3)$$

The content of sum in equation 2.2 can then be understood as a measure of cosine of directions from object centres to the identical point on the surface and therefore a measure of rotation of the meshes.

In 3D space, the result is 3×3 matrix.

The final rotation for single algorithm iteration is then obtained from 4×4 matrix $Q(\Sigma_{px})$:

$$Q(\Sigma_{px}) = \begin{bmatrix} tr(\Sigma_{px}) & \Delta_T \\ \Delta & \Sigma_{px} + \Sigma_{px}^T - tr(\Sigma_{px}) \boldsymbol{I_3} \end{bmatrix} \quad (2.4)$$

where $tr(\Sigma_{px})$ is trace of matrix given by well known formula

$$tr(\boldsymbol{A}) = \sum_{i}^{N} a_{ii} \quad (2.5)$$

Δ is substitution cyclic components of matrix $\boldsymbol{A} = \Sigma_{px} - \Sigma_{px}^T$, therefore $\Delta = [A_{23}, A_{31}, A_{12}]$.

Eigenvector for maximum Eigenvalue is then found and its 4 components form quaternion $\overrightarrow{q_k}$ for the current rotation in step k.

The correspondent translation $\overrightarrow{t_k}$ is found simply from mutual positions of centres. The centre of P_k rotated by $\overrightarrow{q_k}$ must be used to get valid translation for

combined transformation. As the centre of mass equation 2.3 is linear, the centre of rotated mesh is identical to rotated centre of original mesh:

$$\overrightarrow{\mu}(\boldsymbol{R}(\overrightarrow{q}) \cdot A) = \boldsymbol{R}(\overrightarrow{q}) \cdot \overrightarrow{\mu_A} \tag{2.6}$$

where $\boldsymbol{R}(\overrightarrow{q})$ is rotation matrix for quaternion $\overrightarrow{q}$ defined as [4]

$$\boldsymbol{R}(\overrightarrow{q}) = \begin{bmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_1q_2 - q_0q_3) & 2(q_1q_2 + q_0q_2) \\ 2(q_1q_2 + q_0q_3) & q_0^2 + q_2^2 - q_1^2 - q_3^2 & 2(q_2q_3 - q_0q_1) \\ 2(q_1q_3 - q_0q_2) & 2(q_2q_3 + q_0q_1) & q_0^2 + q_3^2 - q_1^2 - q_2^2 \end{bmatrix} \tag{2.7}$$

Then the translation part of transformation from P_k to P_{k+1} is given by

$$\overrightarrow{t_k} = \overrightarrow{\mu_X} - \boldsymbol{R}(\overrightarrow{q_k}) \cdot \overrightarrow{\mu_{Pk}} \tag{2.8}$$

In **step 3** of *ICP* algorithm, the mesh P_k is transformed to P_{k+1} using both rotation quaternion $\overrightarrow{q_k}$ and translation vector $\overrightarrow{t_k}$. For each vertex $\overrightarrow{p_{i,k}}$, new vertex $\overrightarrow{p_{i,k+1}}$ is then obtained as

$$\overrightarrow{p_{i,k+1}} = \boldsymbol{R}(\overrightarrow{q_k}) \cdot \overrightarrow{p_{i,k}} + \overrightarrow{t_k} \tag{2.9}$$

As both rotation and translation are rigid transformations, the shapes of P_k and P_{k+1} remain unchanged. Therefore, if the original mesh P is only similar to the final mesh X, perfect registration will never be found.

Therefore the change of error is used as a stop condition in **step 4** instead of its absolute value. The error is defined as sum of squares of mutual distances between paired vertices of mesh P_{k+1} and X

$$d_k = \frac{1}{N_p} \sum_{i=0}^{N_p} |\overrightarrow{p}_{i,k+1} - \overrightarrow{x_i}|^2 \tag{2.10}$$

This value depends on the scale of meshes as well, so to get general measure of error, [4] suggests using normalised value

$$d'_k = d_k \sqrt{tr(\Sigma_{px})} \tag{2.11}$$

The proof in [4] shows, that after each step, the d_{k+1} is less or equal to d_k. This ensures the stability of the algorithm.

The complexity of algorithm is based on the number of iterations. Each iteration's complexity is bounded by $O(N_p \cdot N_x)$ matching to fitting vertices to each other by trying all possible combinations in step 1. This can however be improved to $O(logN_x)$ using *k-d trees* [4]. Another option is caching of pairs based on expectancy of continuous transformation [21]. This is much lower time complexity than effort put to minimisation of d_k using general approaches for 7 dimensional mathematical function with argument $\overrightarrow{u} = (\overrightarrow{q}, \overrightarrow{t})$ [4].

The positive feature of the approach is the safety of convergence and large amount of modifications discussed below.

The algorithm could be used for our problem, as it handles the mesh regardless of its topology and therefore can manage to register even non-manifold meshes.

However there are some issues that has to be dealt with. First of all, rigid approach could become a problem if the deformation that is needed to perfectly aligned the source mesh to the target mesh is not small. Then there is another limitation not specifically mentioned for this group of algorithms. It assumes that the initial pose of meshes is quite close, therefore a general position in 3D space given by random choose of object alignment in our case would make the algorithm to fail. The algorithm also has problems with meshes without distinctive axes, such as spheres with minor irregularities on surface. In that case, solution is found, but number of iterations can get very big [4]. Another problem is, that the algorithm ensures that a minimal distance transformation is found, but does not specify that the minimum is global. This means that local minimum can be found instead. As the error is not allowed to grow, algorithm is not able to overcome such local minimum and ends. This means, that results can be dependent on initial position of meshes and could be problematic in our application where we do not state any such assumptions about inputs.

Thankfully, we can expect that the main portion of rigid deformation in our data is given by data segmentation errors and differences in the body proportions of individual subjects. Therefore the main non-rigid transformation could be described by scaling so the residual difference should become small enough that adapted *ICP* method would be able to do the registration with reasonable error. We can also make the initial position of meshes close enough for *ICP* to work if we find some rough method for approximate registration before the *ICP* run itself. As for the distinctive shape and local minima problem, we could expect that no such special case would occur that would make these weaknesses to rise as our data do not seem to feature any problematic properties.

Therefore the algorithm might be good choice.

Modifications

Changes in algorithm steps The paper [21] divides possible modifications of the *ICP* algorithm steps into six groups. It also includes a comparison of such adjustments gained by experiments with a single reference implementation and three different test meshes with known solutions.

First possibility is to change the **selection of points**. In the original approach from [4] all vertices from both meshes were considered in all steps of the algorithm. This can however be quite expensive for very large meshes, although extra vertices do not usually add much information on smooth parts of the surface. It is therefore possible to select just some smaller set of points that is representative enough to describe the orientation of the whole mesh.

Then the question is how to select those significant vertices. One option is an uniform selection of points on the mesh. Another is a random pick of points. Alternatively points with distinctive features, such as a large gradient, can be preferred as they are more specific for the shape. The article even suggests one new method based on an uniform distribution of vertex normals instead of positions. Authors expect this approach to better cover characteristic of the mesh. This seems to be vital for meshes with bad normal distribution where all characteristic vertices are concentrated in small area thus leaving the traditional uniform approach useless.

Proper point selection can maintain reasonable registration error with lower computation price. Normal uniform and random sampling seems to be the most reliable solution for general meshes. However our solution does not require real time performance, therefore sticking with all points sampling would be the obvious safest choice.

The random pick in some of the methods adds another interesting feature. If point selection varies between iterations, some small local minimum can be overcome. On the other hand, the convergence is no longer certain as some pick of points may lead to different solution than other.

The quality of sample selection can also be altered by taking both meshes P and X into account, so that points are not only projected from P to X but also vice-versa [21].

Traditional approach [4] then used Euclidean **metric** to pair vertices from both meshes. Other options from [21] are based on projection of point from P to the mesh X and locating nearest vertex of incident triangle. This can be either done in direction of normal or in direction of view ray from target mesh X perspective.

Limitation can also be added so that paired vertices must share some additional quality to specified degree. This can be either geometry information such as normal direction or additional information such as colour or density [21].

Results show that the original closest point approach is by far slowest even with $O(logn)$ search structures. It is even more significant when the execution time is considered instead of iteration count. However when it comes to meshes with low amount of distinctive features, projection based metrics to minimise the error and therefore to get ideal registration. The closest point metric shows to be more robust [21]. In our case, the calculation time is not as important as the robustness of the metric.

In equation 2.2, the cross-covariance matrix is obtained from products of vertex pairs as a normal sum. This is equivalent to uniform **weighting** of all pairs. [21] identifies the change of such weights as another possibility to enhance algorithms performance.

Weights can be chosen based on vertex selection approaches discussed above [21]. Therefore more distant pairs can have lower weight assuming that the pair may be false. Or the difference of normal angles determined measured by dot product can be extension to angle difference threshold. Other choices are more individual and build upon more knowledge about data origin. Such an example is known precision of scanner based on camera position [21].

Results show, that this modification has only small general impact on algorithm qualities.

For our problem however, these weights might reduce problems with non-manifold area vertices by assigning lower values to problematic parts of meshes where the probability of mesh errors is higher. The question then would be how to find and measure such parts.

In extreme variant of zero weights for some pairs, the weighting leads to the complete **pair rejection** which is the fifth modification group in [21]. This causes rejection of pairs beyond some threshold of specified metric such as usual point distance, normal angle difference or inconsistency on mesh. Inconsistency is a measure of point surroundings similarity oh both P and X meshes, determined for example by distance between pair point and its neighbours. This however assumes that both meshes have similar qualities such as a vertex count and a distribution, which is not our case.

However, [21] shows that these adjustments do not bring any significant benefit for general meshes.

Last step in *ICP* iteration and last part to modify is error measurement. Originally it is quantified using sum of squares of point-to-point distances (see eq. 2.10), but it can as well be expressed by distances from the point to the nearest triangle plane in the other mesh.

This allows usage of different minimisation techniques discussed below which leads to faster convergence [21].

Acceleration The original paper [4] suggests acceleration using polynomial approximation of last three errors. This allows a better prediction of new transformations in the current step k based on transformation vector $\overrightarrow{u_k} = (\overrightarrow{q_k}, \overrightarrow{t_k})$ calculated traditional way in this iteration and transformations $\overrightarrow{u_{k-1}}$ and $\overrightarrow{u_{k-2}}$ from previous iterations. Final $\overrightarrow{u_k}$ is than obtained as a prediction based on their change.

This does not change characteristics of the algorithm, but leads to a reduction of the number of iterations.

Approach was further developed by authors of [21]. Authors state that their modifications were able to reduce overshoot of extrapolation. This allowed them to create real-time implementation of *ICP* for scanner image processing.

2.2 Non-rigid methods

Non-rigid methods do not rely on the fact, that the meshes represent the same object in the same position. Therefore they have to transform not only complete mesh but individual vertices as well to achieve non-rigid transformation and compensate initial deformation. Differentiation of these methods is based on amount of deformation they are able to handle. While the first group expects rather small errors caused by technical properties or imperfections in samples, the other one aims for registration of fully deformable objects.

2.2.1 Small deformations

Small deformations in this context are not usually deliberate deformations at all. They are often caused by different scanning angles, various techniques or changes in the scanned object. Therefore only small errors that could otherwise be ignored by rigid methods are handled. The result of this should be better quality of final registration.

An example of such method is described in article [5]. Aim of this article was to improve registration of different view scans and to keep more high frequency details in the final model.

The method works with group of 3D meshes at once and assumes that it has approximate alignment in the beginning. This comes from the scanning process itself but could be gained by some feature based method above as well. I will first describe a configuration with only two meshes.

First traditional *ICP* is done to match the source mesh to the target one. If the error is too big, two meshes does not overlap enough and the algorithm stop.

Then feature points are selected in both meshes. Only small number (1% or even less) of points is used. Each feature point of each mesh is again target of *ICP* to find its position on the other mesh independently. This means choosing some other points in feature point's surrounding that will be aligned by this new *ICP*.

Selection of these neighbours is done randomly and probability of choice has two criteria. First one is distance from main feature point described as [5]

$$p_{feature}(\overrightarrow{x}) = \frac{1}{\epsilon + \| \overrightarrow{x} - \overrightarrow{f_i} \|^2} \tag{2.12}$$

where $\overrightarrow{f_i}$ is feature point and $\overrightarrow{x}$ his neighbour. Therefore nearer points on same mesh are preferred.

Second criterion is expression power of point selection. To have such, points must lie on some geometrical distinctive part, not plane for example. Therefore another function based on local normal difference is stated [5]:

$$p_{stability}(\overrightarrow{x}) = (\overrightarrow{x} \times \overrightarrow{n_x}, \overrightarrow{n_x}) C^{-1} \begin{pmatrix} \overrightarrow{x} \times \overrightarrow{n_x} \\ \overrightarrow{n_x} \end{pmatrix} \tag{2.13}$$

where C is covariance matrix of overlap area.

This prefers points with normal aiming to different direction than rest of selection.

Final probability of selection of point x is then

$$p(\overrightarrow{x}) = p_{feature}(\overrightarrow{x}) \cdot p_{stability}(\overrightarrow{x}) \tag{2.14}$$

Feature points are therefore selected on both meshes and then conventional *ICP* is done for each feature point of both meshes targeting the second mesh. This means many calls of iterative algorithm. Authors of [5] however claim, that thanks to previous close alignment of meshes, all *ICP* will be very fast and will end after few iterations.

Now, we have many feature points and their images in the other mesh. All these images have to be somehow combined together.

First, feature points that were accidentally selected to lie too close to each other are pruned. Same applies to feature points that are significantly further from their images than other feature points in their surroundings. This should remove outliers as article says.

Error energy equation is then stated to tie each feature point to feature points in his surroundings:

$$E(\overrightarrow{g_i}, \overrightarrow{g_j}) = \sum_{P_k} w_{ij} \left(|\overrightarrow{g_i} - \overrightarrow{g_j}| - |\overrightarrow{f_i} - \overrightarrow{f_j}| \right)^2 \tag{2.15}$$

where $\overrightarrow{g_i}$ are final positions for respective feature points and weights w_{ij} are probably based on distance between points although this is not mentioned in the article.

Minimising such energy for all pairs of feature points means that distance between two feature points on both mesh and in final positions should remain unchanged. The minimum is found in a least square manner using simple gradient descent method. Figure 2.1 shows effect of such approach on feature points without any matches that would be left in their initial position otherwise.

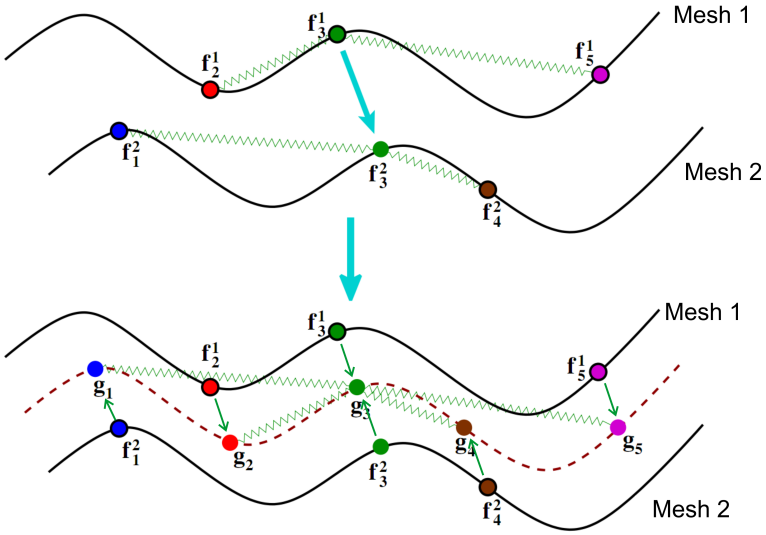


Figure 2.1: Finding final positions g_i for feature points g_i in registration method from [5]. Note that points without image on the other mesh are forced to proper position by their neighbours. Taken from [5] and edited.

Now both meshes must be independently warped to match its feature points to final positions.

Interpolating thin-plate splines are used to describe both initial feature points $\overrightarrow{f_i}$ and final positions $\overrightarrow{g_i}$. Non-feature points on mesh are then interpolated based on their position on spline.

If more than one mesh like in the original paper are on input, all pairs of two meshes are registered by initial *ICP*. Too distant pairs are then pruned. The

remaining pairs of meshes are targets and sources for the registration of the feature points.

The authors of the paper then presents that the method is able to prevent smoothing and artifacts on final models made by scanning. With Michelangelo's David having 28 million vertices, tens of hours on computing cluster were required. Smaller models were registered in tens of minutes. This means that it is not real time algorithm but this is of no concern for us as we do not require that.

This method has nice resemblance to our problem in the number of input meshes processed at once. The difference is that our meshes are mostly complete and although some parts might be cut off to remove non-manifolds, there will still be enough overlap between each of them. This would make the algorithm even simpler.

The ability to fix minor deformations suits our needs well as our models come from various scanning techniques and various subjects. Problem however remains, that pre-processing would have to be performed to ensure initial alignment in a global space and also to cope with varying model scales.

Our meshes are also more different from each other as they are not from the same scanner as in the paper. We can therefore expect, that there will be larger distances between rigidly aligned meshes that might not be fully fixed by suggested region selection approach.

2.2.2 Large deformations

This group of methods do not assume that surfaces have at least nearly same shapes. It therefore allows large deformations in the mesh.

An example of such method can be found in [11]. Although both meshes do not have to be close to each other on most of their surface, there is still an assumption that some part of the inputs is overlapping and therefore the meshes are not in general initial position. Then combination of iterative optimisation process with filtering of false relations leads to registration across the mesh surface. The main issue is dependence on topological similarity of both meshes that cannot be guaranteed for our inputs. An additional problem is the requirement of good initial alignment in at least small part of the mesh. This is easier to ensure if we know the source of deformation but not so easy if two different meshes without common origin are registered.

2.3 Summary

Relevant methods for mesh to mesh registration have been described. This list is not exhaustive as only some of the distinctive ones have been picked to show variety of approaches. Extended version of this introduction can be found in Master Thesis [14]. Extra information can also be found in citation lists in referenced articles can be used. Especially [21], [5] and [11] contains links to many other examples in the introduction parts.

From our perspective, important features of algorithm is ability to handle non-rigid objects, global registration with no previous alignment and potentially incomplete meshes with varying topology. Our demand for non-rigidity support was best matched by method from [5] as it is able to perform local deformations for better alignment of meshes with minor deformations. This could solve disproportional errors in our input meshes caused by various subject origin and different segmentation. This however may not be necessary if those differences show to be small enough. General *ICP* method could then be used instead allowing simpler implementation and performance benefit as well.

Most of the *ICP* methods I described assume some sort of initial alignment. For some of them it is fundamental as they would fail otherwise ([11]), with the others the risk of finding false local minimum growths with the initial distance in both translation and rotation.

Another way of fixing problem of initial assumption is to provide such needed alignment. I will discuss usage of *principal component analysis* [12] later in Chapter 4.3.1.

The last demand of non-rigid or incomplete mesh support is easier to fulfil as most of the methods aims for registration of 3D scans, which are partial and registered to create complete model. Therefore we should be able to perform registration even if we were forced to cut some parts of input data out for their corruption, e.g. non-manifold edges.

To sum it up, I will try to use rough alignment of meshes before the full vertex-per-vertex registration. If this shows to be too approximate, I would prefer an *ICP* based method such as from [5], as it is able to perform registration on locally deformed meshes.

Chapter 3

Multi-morphing of surface meshes

In the previous chapter, the topic of mesh registration was discussed. While the registration is process that somehow adapts one mesh to match the other, we might also want to keep features of both input meshes and produce a result mesh that would somehow mix both inputs up. This way, there will be no source mesh to deform nor target one to be approximated, but only two or generally multiple input meshes and the algorithm should produce single new mesh as combination of all of them. Then such concept is described by the term *morphing*. We will need it to produce the final mesh, while the previously described registration method will work as a preprocessing that tells the morphing how to map each mesh to the other ones. Simply said, the registration prevents mixing up head and leg if two people are morphed into one.

Morphing in computer geometry is a process of interpolation from one entity to another. Those entities can be anything from raster images such as photos through 2D shapes like polygons to 3D or higher dimensional objects. For my work, 3D mesh objects are the main interest.

Morphing can be used in animations, to allow smooth transition from one model to another. Typical case is metamorphosis of gaming character. If original models are identical object in different pose, movement can be animated using such technique. In our case, we will use several models of the same object in the similar pose but with different representation and quality to get new model with better quality.

As we do not limit our aims to two meshes only, we speak about multi-morphing. Most of the principles are however same as in two mesh case and therefore following descriptions will mostly cover the simpler case.

3.1 Basic concept

Common ideas and key factors for mesh morphing are described in doctoral thesis [20]. I will first summarise those basics and then look little deeper to individual parts in context of my goal.

The main problem of mesh morphing comes from different topologies of input meshes. As the number of vertices of two meshes is generally different, there is no bidirectional mapping between them. If there was, then simple interpolation between such pairs would solve the problem.

Therefore general solution is to represent both meshes in a same way. This is done by projection to *parametric domain*. This can be plane for some unclosed meshes, but mostly it is sphere for genus 0 meshes. Those are meshes that can be deformed into sphere without cutting. We can say, that our meshes will satisfy this condition. There are also higher genus domains, such as torus for genus 1 ([20]), but we shall not need those. It is also sometimes possible to project even genus 0 mesh into plane, but it involves cutting and brings unnecessary complications to later phases [27] (see Section 3.6.1 for insight).

Now we have spheres for both meshes such as that each vertex of sphere presents parametrisation of single vertex on original mesh. This means that there is bidirectional mapping between the mesh and its parametrisation.

Next phase is therefore to find relations between both meshes. This is done on parametric domain, sphere. Now sphere parameter $\overrightarrow{p_i}^P$ of source mesh's P vertex p_i is expressed in barycentric coordinates of parametric triangle j of mesh X where it lies when both spheres are aligned (see Figure 3.1). This means that each parametric point $\overrightarrow{p_i}^P$ is expressed in terms of mesh X only as

$$\overrightarrow{p_i}^P = \alpha\overrightarrow{x_{j0}}^P + \beta\overrightarrow{x_{j1}}^P + \gamma\overrightarrow{x_{j2}}^P \tag{3.1}$$

In the simplest case, the relation could now be projected back to original meshes, thanks to bidirectional mapping of parametric and original vertices. This would analogically lead to representation of each vertex $\overrightarrow{p_i}^X$ in term of mesh X

$$\overrightarrow{p_i}^X = \alpha\overrightarrow{x_{j0}} + \beta\overrightarrow{x_{j1}} + \gamma\overrightarrow{x_{j2}} \tag{3.2}$$

The superscript X in $\overrightarrow{p_i}^X$ denotes, that in general $\overrightarrow{p_i} \neq \overrightarrow{p_i}^X$. $\overrightarrow{p_i}^X$ represents position of point on surface of target mesh X that is nearest match for point $\overrightarrow{p_i}$ in source mesh P. Therefore simple method would then only interpolate those two points to achieve morphing between mesh P and X maintaining topology of mesh P. This also means that shape mesh X would never be achieved perfectly.

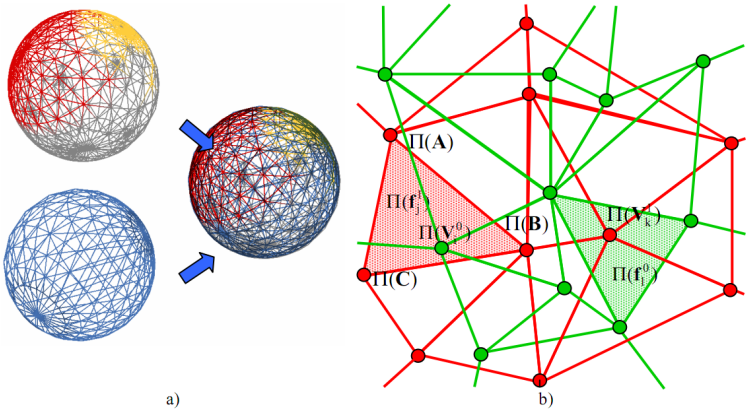


Figure 3.1: a) An example of parametrisation on spherical domain of two meshes. b) Detail look into aligned sphere surface where barycentric coordinates of vertices of one mesh are found in second one. Taken from [20].

More complicated solution uses one extra step called *remeshing* [20]. This is general idea of constructing common *supermesh* which contains features from both meshes P and X. Then vertices of such mesh are expressed using equations 3.1 and 3.2 instead. This mesh then also creates output. As it can have more vertices than both meshes P and X, it can sustain details of each of them in both extremes.

3.2 Parametrisation

Parametrisation is one of two more complicated parts of general algorithm described in the previous chapter. In our case of genus 0 mesh, we are trying to expand the surface to sphere. If there was an inside point, from which every vertex of mesh was visible, one could just project the mesh to sphere by normalising directions to each vertex from this central point. They are some meshes like this and they are called star-shaped as cartoon style star is an example of such shape.

Unfortunately most of meshes lack such property and therefore overlaps occur if projection to sphere is used (see Figure 3.2). Article [1] describes method that fixes those problems and enables spherical parametrisation of general modus 0 meshes.

It starts with simple sphere projection from inner point of an object. Then relaxation follows as iterative process. It penalises long edges to equalise distribution of mesh vertices and prevent collapse to single point. Therefore for each parametric image $\overrightarrow{p_i}^P$ of original vertex $\overrightarrow{p_i}$ penalty vector $\overrightarrow{e_i}$ is calculated [1]:

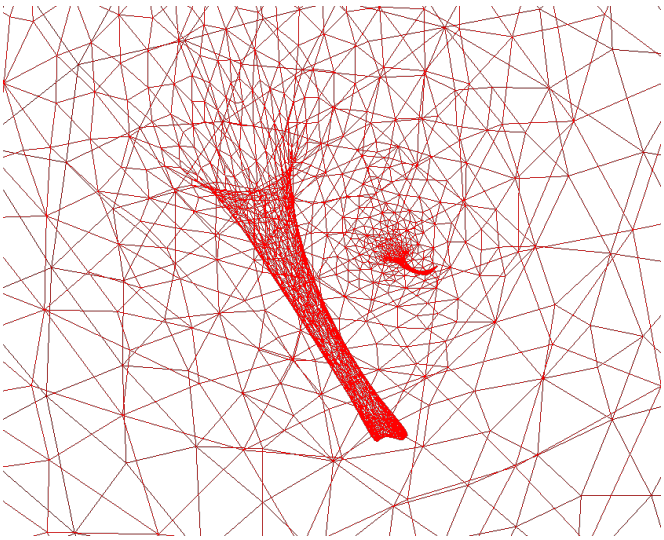


Figure 3.2: Overlap in parametrisation of non-star mesh.

$$e(\overrightarrow{p_i}^P) = c \cdot \frac{1}{|N(\overrightarrow{p_i}^P)|} \sum_{\overrightarrow{v} \in N(\overrightarrow{p_i}^P)} \left((\overrightarrow{v} - \overrightarrow{p_i}^P) \cdot |\overrightarrow{v} - \overrightarrow{p_i}^P| \right) \tag{3.3}$$

where $N(\overrightarrow{p_i}^P)$ denotes set of direct neighbours of vertex $\overrightarrow{p_i}^P$. Constant c says how much long edges are penalised. Article states that it should match inverse of longest edge of penalised vertex.

This penalty is then subtracted to vertex position and result is normalised:

$$\overrightarrow{\widetilde{p_i}}^P = \frac{\overrightarrow{p_i}^P - e(\overrightarrow{p_i}^P)}{|\overrightarrow{p_i}^P - e(\overrightarrow{p_i}^P)|} \tag{3.4}$$

That ensures that parametric coordinates still lies on unit sphere.

Original text contains mistake in the equation above. It mentions that result of subtraction should be normalised, but the formula itself contains addition instead. I have verified the proper version using my own implementation as well as by implementation of Ing. J. Parus, Ph.D. (see Section B.1.4).

Important property of good parametrisation is that no triangles overlap. This is true when all triangles have same orientation [1] and therefore stop condition is based on test of all triangles one by one. Orientation of triangle with vertices $\overrightarrow{a}$, $\overrightarrow{b}$, $\overrightarrow{c}$ is simply calculated like oriented volume of induced tetrahedra[1]:

[1] Volume of tetrahedron would be multiplied by $\frac{1}{6}$.

$$sign(T_{a,b,c}) = sign(\overrightarrow{a} \times \overrightarrow{b}) \cdot \overrightarrow{c} \tag{3.5}$$

An obvious assumption is that original triangles have consistent clockwise or counter-clockwise definition. This way parametrisation without overlaps is achieved for general modus 0 mesh. This is important for the barycentric coordinates as it ensures that they can be found unambiguously for any point on surface of the sphere domain.

3.3 Supermesh construction

As was mentioned in basic description we can use some of the input meshes topology as the output one. That might be enough if the selected input has high number of polygons but in other cases it might limit the number of features that can be expressed. Alternative approach is creating one new mesh topology that would be deformed by the morphing instead of the inputs and used as the output later. Such artificially created mesh is referenced as supermesh in the [20]. Supermesh can be created to contain all features of both input meshes and therefore maintain edges and vertices if interpolation goes close to one or the other of them.

An example of algorithm for construction of such supermesh is described in [1] and [20]. Both these approaches are very similar as they are inspired by [15]. I will therefore present an example based on the newer one of them described in [20].

The supermesh construction there is based on insertion of edges from target mesh to source mesh. The process starts on parametric domains instead of original meshes as it makes it easy to track path of edge on different mesh. Therefore each edge from parametrisation of target mesh X is taken and inserted into parametrisation of source mesh P (see Figure 3.3a). Then intersected edges are found and intersection vertices are inserted (3.3b). Triangle walking is suggested to optimise complexity of intersected edge location. Edge to edge intersection test as well as point to edge position test are potential source of numerical instability [21].

Created mesh is not triangular, therefore triangulation is the next step (3.3c).

Triangulation can be done either after each edge insertion or at the end. The first case is easier ([20]) but unnecessary edges can be inserted as they would otherwise be provided from target mesh in following steps.

The second case inspects the supermesh on per vertex basis. Sorted fan around each vertex is built and each pair of consequent neighbours is checked for mutual

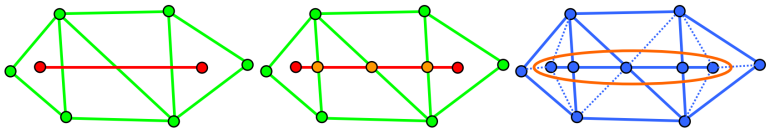


Figure 3.3: a) Orange edge from parametric representation of target mesh X inserted to parametric representation of source mesh P. b) Intersection points are inserted. c) Triangulation is fixed. Taken from [20].

connectivity. If none edge between them exists, then new is created resulting in creation of new triangle.

At the end, supermesh is projected to source mesh and target mesh using barycentric coordinates as described in 3.1. Interpolation between those two meshes and their vertex positions is the key to the morphing.

There are some additional enhancements described in [20]. First suggestion is that new vertices could be inserted to Bezier splines instead of flat triangle faces, keeping the surface more smooth based on normal values in surrounding original vertices.

Then there is an idea to improve supermesh triangle quality by flipping of some of extra edges added in triangulation step, so they for example obey Delaunay condition. This would make some later computations more stable.

Final supermesh can also have unnecessary amount of triangles. The author states that this can be fixed by reducing number of edges added in merging phase. He says that edges between nearly parallel triangles do not carry much information about shape and can be left out.

The last idea is most relevant to me as it describes merging of multiple meshes. The modification is very simple. Meshes are just paired and processed in divide-and-conquer manner so that only $logn$ merging operations are performed instead of n.

Our meshes usually contain enough vertices and not many sharp edges that could cause most severe artifacts if bad interpolation mesh is used. This means that there is a chance, that supermesh creation could be avoided. If this assumption shows wrong, I would use the description above to build a new mesh. This may be possible to happen if the manifolds areas cut off in pre-processing are large.

Alternative representation is also mentioned in [20]. It says that instead of ordinary vertex positions, for example Laplacian vectors of individual vertices can be interpolated. This representation is based on difference of vertex and weighted average of neighbours and therefore describes local shape of mesh invariant on position in space. This way, two objects can be interpolated even if they lie in different places in space.

3.4 Interpolation

I will not go deep into the last part of morphing pipeline as the easiest and most common choice of linear interpolation should be sufficient for our needs.

However for cases where two meshes describe completely different object, such as two animals, higher-order interpolations may produce smoother results. For example, Bézier interpolation polynoms can be easily expressed, if source and target vertex normals are used as spline tangents [20]. This also means that not every vertex must have the same interpolation ratio in single pose. We might use that idea to minimise influence of incorrect meshes in problematic parts.

3.5 Recent development

I have tried to find a more recent developments in the mesh morphing techniques. The paper [27] describes method for morphing of two general genus 0 meshes. As well as in [1], these objects does not have to be star based as can be seen in Figure 3.4. This is made possible by incorporation of relaxation schema that fixes possible triangle overlaps.

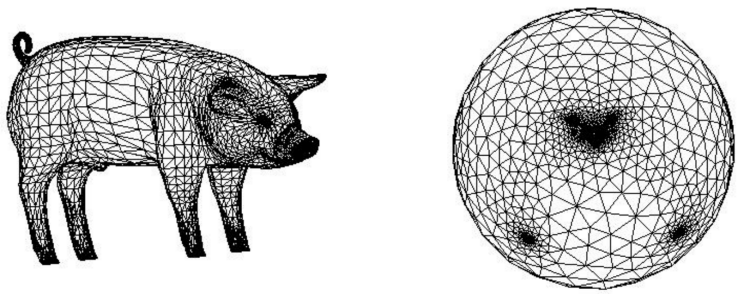


Figure 3.4: Genus 0 mesh of pig and its spherical parametrisation. Note that even that mesh is far from being star based, parametrisation still avoids overlaps. Taken from [27].

Let us describe the method in more detail. It first makes rough alignment of meshes using principal component analysis. This finds main orthogonal axes which are then aligned with main x, y, z axis of coordinate system.

Next step is sphere domain parametrisation. Authors suggest very simple, not extremely fast but, judged by the results, a functional solution for fixing overlaps.

Projection to sphere is the first step of parametrisation. Then relaxing energy for each parametric vertex $\overrightarrow{p_i}^P$ is expressed as arithmetic average of its direct neighbours [27]:

$$e(\overrightarrow{p_i}^P) = \frac{1}{|N(\overrightarrow{p_i}^P)|} \sum_{\overrightarrow{v} \in N(\overrightarrow{p_i}^P)} \overrightarrow{v} \tag{3.6}$$

where $N(\overrightarrow{v})$ is set of direct neighbours of vertex $\overrightarrow{v}$. To maintain position of sphere, $e(\overrightarrow{p_i}^P)$ is normalised. Then for next iteration, each parametric coordinates for each vertex are simply put to be

$$\overrightarrow{\widetilde{p_i}}^P = \frac{e(\overrightarrow{p_i}^P)}{|e(\overrightarrow{p_i}^P)|} \tag{3.7}$$

Although the stop condition is not mentioned in the article, a similar approach as in [1] can be expected. That means that iterations stop when all triangles has the same face orientation.

As the method aims for morphing between different objects, features must be paired to lie in same positions of parametric domain. This is done manually by user, who specifies that nose of bear is equivalent to nose of tiger and so on. Then matching parametrisations of such vertices are replaced by their mutual averages, so that they lie in the same place. This creates overlaps in the parametric mesh and therefore new relaxation is run with fixed positions of selected feature points.

Last phase is simple as authors do not use a supermesh and only interpolates one mesh from its initial form to representation on surface of target mesh. This means that vertices of one mesh are matched to the linear combination of vertices of the target mesh using barycentric coordinates of spherical parametric projection of each source vertex to corresponding spherical triangle of target mesh, as was already described in Chapter 3.1 with all necessary equations. Linear interpolation is used in the last phase.

The algorithm is very similar to approach from [1]. The only difference I was able to spot was change in the parametrisation relaxation process. Overall this method seems to be much simpler than the one from [1] as it handles the parametrisation features alignment in more transparent way and it also does not involve construction of supermesh. The article does not contain direct comparison so I would only speculate that this can lead to worse performance if meshes are less similar in topology. Simplicity of relaxation condition could also mark higher iteration counts.

Results show that for meshes such as in Figure 3.4 a smooth interpolation is achieved. These shapes are even more complicated than those we are expecting to deal with. This means that parametrisation approach described should be possible to use.

There might be problem with proper alignment of meshes without user specified feature point as our method should be automatic. However, there are two assumptions I have. First in our case only very similar meshes are morphed and therefore the parametrisation should be aligned properly. If this show to be incorrect, similar approach as in [27] could be used with feature points randomly sampled from mesh registered by some non-rigid method such as [5] described in Chapter 2.2.1.

Article presents that times under one second are achieved for meshes with few thousands vertices. This is more than sufficient performance for our non-real time needs. We could therefore benefit from simplicity of parametrisation approach without any significant loss caused by its possible inefficiency compared to more complex variants.

3.6 Multi-morphing

Multi-morphing is generalisation of morphing to space of any number of base meshes.

Main condition for multi-morphing is that all base meshes have same size and connectivity. This is solved by supermeshes discussed in Section 3.3. The supermesh construction was done on pair of meshes but in the later part, possibility of multi-mesh merging was discussed and was realised on pair by pair basis.

However, [20] states that such supermesh has poor quality and excessive number of triangles. This is easy to see as every merge and insertion of edges from individual base mesh or partial supermesh adds more and more often duplicate or expendable edges that are no use for the final output. Even after application of enhancements discussed earlier, such as Delaunay retriangularisation and omitting of auxiliary edges, final mesh might be very bad if there is high number of high polygonal inputs in the beginning.

3.6.1 Adapted supermesh

There are different approaches more optimised for multiple-mesh merging. An example is method described in article [18].

The algorithm is however again presented on simplest dual mesh case. The inputs are two meshes of general genus value and set of user given feature vertices with correspondences. Those are positions of nose, limbs and so on. First, common very rough mesh is constructed using these features from both input meshes so that edges of this coarse mesh are formed from original edges of input meshes.

This is done by choosing of shortest paths between feature vertices satisfying condition that they maintain mutual position of other feature vertices and other chosen paths. This way, both meshes are cut into *patch layouts* that after replacement of paths by edges forms two new low polygonal meshes with identical topology (see Figure 3.5).

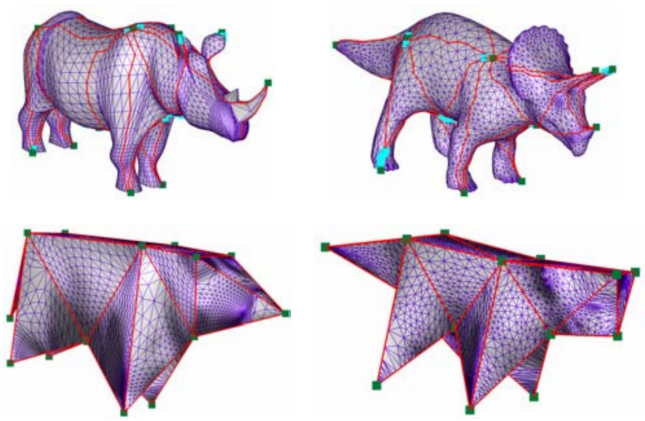


Figure 3.5: Input meshes with selected paths between feature vertices forming triangular patches (top). Base domain mesh with common topology created from those patches (bottom). Taken from [18].

Sometimes patches are not triangular, meaning that there are not exactly three feature vertices on their boundaries. Then face paths are found between feature vertices that should be connected. Face paths are converted to edge path, inserting new vertices into middle if necessary. This way all patches are "triangulated".

To improve quality of patches in the triangle mesh sense, some "edges" (paths) are flipped based on degrees of patch vertices (feature vertices).

Authors of [18] warn that even after edge flipping, quality of domain meshes may be poor if straight paths from input mesh are converted to straight edges. Therefore they perform iterative relaxation of input vertex positions resulting to change of domain triangle where the vertex belongs.

Domain meshes are then used as parametric domains for their input meshes. It is even simpler than in spherical domain case, as we know which input mesh vertex lies in which patch that matches to single domain mesh triangle. Therefore the parametrisation is done on plane for each vertex. This is an example of genus 0 (or higher) mesh being parametrised on plane mentioned in Section 3.1.

Therefore bijective mapping between input meshes and their domains is found and then mutual mapping between domain meshes is simply observed from the fact that they are isomorphic meshes. This then allows indirect mapping between both input meshes as described in the introduction part of this chapter.

To build a supermesh, one of the input meshes is selected to be a base. It is

probably best to pick the one with the best quality of triangles and with the highest resolution to cover the features of other meshes. Then the second mesh is projected to the first one. Until here, this is the simplest possible morphing approach without supermesh discussed at all. However, the quality of approximation for other models may be poor in some complicated parts, such as ears of cow in the original article.

Authors of [18] fix this in similar way as they improved patches in individual meshes. They run iterative relaxation algorithm and they penalise edges based on their error, error in the middle point and also error in previous iteration for better stability. The error itself is difference between original vertex of other mesh and reconstruction of its image based on back projection from supermesh.

This way, vertices of supermesh are redistributed to better cover both models. However sometimes this is not enough and some vertices are added to areas where relaxation does not achieve to minimise error under threshold. This is done by edge splitting.

Last phase fixes normal directions. Authors avoid edge flipping to keep the supermesh topologically compatible with its starting mesh. Hence they just split those edges where normals should be corrected instead.

Final supermesh is superset of graph of its source mesh and it is also good approximation of the second input mesh. As algorithm prefers relaxation and position adjustment of vertices already existing in the supermesh to insertion of new vertices, resulting supermesh can be significantly smaller than that produced by method described in Section 3.3 ([18]).

This is illustrated on results showing that the supermesh vertex count is limited to something about 50% of size of the largest input. If the other algorithm performs edge intersection for almost all edges, it must unavoidably produce much larger meshes. This means smaller triangles and larger numerical errors in later computations.

Until now, I have spoken about two input meshes only, although the benefit of approach should have been in ability to process multiple meshes and maintain quality of produced supermesh. Authors of [18] states that the most efficient way of generalisation for more inputs is selection of one template mesh and then finding its correspondences, patches and domain meshes with all other meshes one by one. This can obviously be done parallel. Then the construction of supermesh is iterative, so that supermesh created from template mesh and other mesh i is new template for supermesh created with next other mesh $i + 1$.

Main problem of this approach is that it requires user defined feature vertices. Those are chosen semantically and are intuitively distributed to cover the surface and to strengthen important parts. If we wanted to adapt this method, we could make the selection of such points on one mesh only and use registration algorithm

to find correspondences on the others.

It also seems obvious that this method is more robust in algorithm complexity and thus harder to implement.

3.6.2 Affine morphing space

Using supermesh, every input mesh can be expressed in the same topology. Therefore graphs off all these meshes are isomorphic. This allows construction of *Affine morphing space* (*AMS*) and *Morphing vector space* (*MVS*) described in [20].

AMS and *MVS* work in the same way as usual affine spaces. Points are replaced by vectors of mesh points, vectors by vectors of vectors. All basic operations such as addition, subtraction, dot product and length calculation can then be applied on individual element level. Simple example - in standard algebra, one dimensional space of line can be described using interpolation between two points $\overrightarrow{a}$ and $\overrightarrow{b}$ as $t \cdot \overrightarrow{a} + (1-t) \cdot \overrightarrow{b}$. In *AMS*, the same can be done for space defined by morphing between meshes A and B with the same topology. Such space would than be described as $t \cdot \overrightarrow{A} + (1-t) \cdot \overrightarrow{B}$. Then these operations denotes application of them to individual pairs of points of both meshes.

Hence n input meshes are considered basis of $n-1$ dimensional space in which other meshes can be expressed by barycentric coordinates or as linear combination of base vertices between first base mesh M_0 and other M_i, $i > 0$. All this maintains perfect consistency with regular point and vector algebra.

This then goes even further, when orthogonal projection is calculated the usual way using dot products and Gram matrix [20], which then gets coordinates of new mesh in the space defined by original base meshes or distance to that space if new mesh does not lie in it. This then allows analysis of another meshes and measurement of their similarities to the bases.

However, this all goes little too far from our needs as our interest in mesh morphing is mainly straightforward. We just need to use basis mesh space to create single new interpolated mesh, which will be output of our program. Therefore we can see all the math behind terms like *AMS* and *MVS* in very intuitive way and just express the new mesh as linear combination of input meshes keeping barycentric coordinate condition, such as the sum of interpolation weights will be 1, so extrapolation is avoided.

3.7 Summary

I have described general approach used for morphing aimed at genus 0 surface mesh models. I have discussed several specific approaches and evaluated their suitability for our problem.

In this phase however it is hard to estimate necessity of usage of complicated approaches like multi-morphing supermesh construction [18] instead of simple methods like using the best mesh as morphed mesh [27].

We also have to consider the benefit of using existing implementation of spherical parametrisation from [1] instead of creating new code for different parametrisation techniques like patch detection in [18].

I will therefore try to construct the final algorithm from the simpler methods, evaluate the results and locate problematic parts that could be improved by different approaches.

Chapter 4

My solution

In this chapter, the method for automatic mesh morphing will be built step by step. Section 4.1 describes the inputs of the algorithm and limitations applied on them. Then each other section adds one major step to the complete algorithm. During the initial experiments, two distinctive variants of the method were created. The first one is referenced as variant with spherical domain parametrisation and the second is referenced as direct on-mesh morphing in the later text.

The spherical domain parametrisation morphing version of the method consists of five major steps. The steps are as follows:

1. Remove various artifacts such as non-manifolds from input meshes. The target state consists of closed manifold meshes. This is the topic of Section 4.2.

2. Find a rigid and approximate initial alignment of meshes. The target state are meshes lying approximately on the same place, heading the same direction and having the same scale. This is described in the following Section 4.3.1.

3. Use that initial alignment as a starting point for some advanced registration. The result should consist of slightly deformed overlapping meshes with minimum surface to surface differences and therefore very good mutual alignment. How to achieve that is discussed in Section 4.4.

4. Use these deformed meshes to find topologically equivalent spherical representations. This will produce spherical meshes usable as parametrisation domains. The algorithm and a modification is presented in Section 4.5.

5. Finally, use spherical domains to find relations between meshes and with that information morph all input meshes into one final result mesh. Details in Section 4.6.

These five blocks matching to respective five sections are connected to one large work-flow diagram in Figure 4.1.

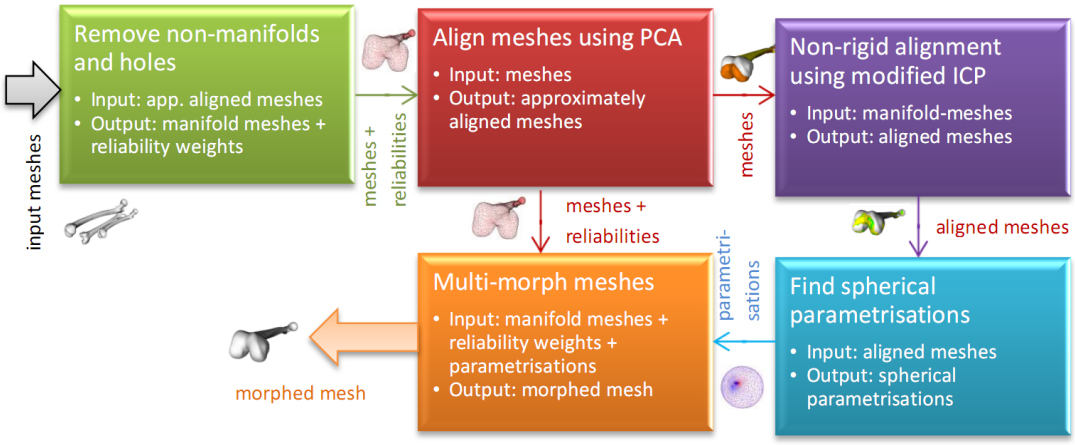


Figure 4.1: Flow diagram of the method variant using spherical domain parametrisation for morphing.

However, the later experiments will show this conventional schema to be unreliable mainly due to the step 4 of parametrisation. This is way in Section 4.7 one extra step is presented that replaces original steps 4 and 5 and builds the alternative **direct on-mesh morphing** version of the method. It has therefore only 4 main steps showed as block in diagram in Figure 4.2. The altered method works as follows (steps 1 to 3 remain unchanged):

1. Remove various artifacts such as non-manifolds from input meshes. The target state are closed, consistent and manifold meshes. This is the topic of Section 4.2.

2. Find a rigid and approximate initial alignment of meshes. The target state are meshes lying approximately on same place, heading same direction and having same scale. This is described in the following Section 4.3.1.

3. Use that initial alignment as starting point for some advanced registration. The result should consist of slightly deformed overlapping meshes with minimum surface to surface differences and therefore very good mutual alignment. How to achieve that is discussed in Section 4.4.

4. Morph the input meshes using mutual relations found directly on the non-rigidly aligned meshes without any special intermediate domain. Produces single final mesh. How to do this is the content of Section 4.2.

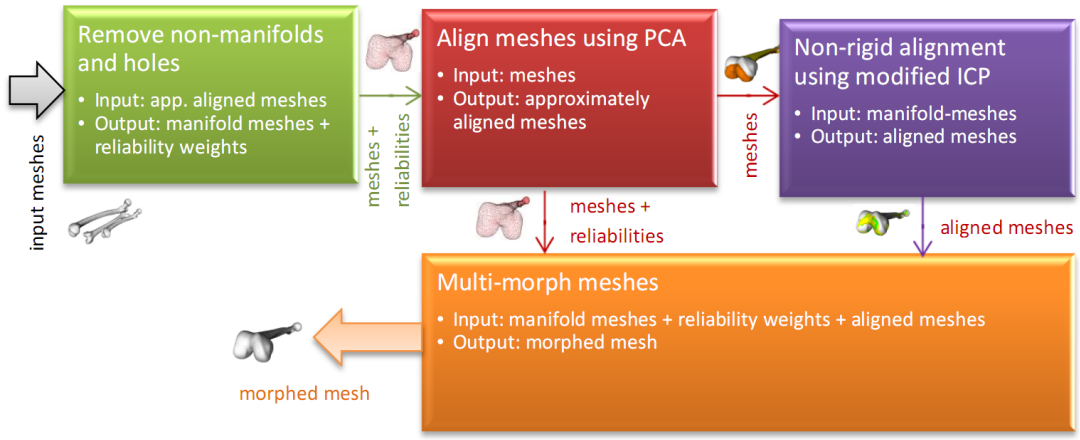


Figure 4.2: Flow diagram of the method variant using direct on-mesh morphing.

To sum it up, Sections 4.2, 4.3.1, 4.4, 4.5 and 4.6 describe five consequent steps of the spherical domain parametrisation version of the method and sections 4.2, 4.3.1, 4.4 and 4.7 four steps of the direct on-mesh morphing alternative.

4.1 Input specification

The input of the algorithm is a multiple mesh set (see Figure 4.3). The input meshes are assumed to be non-manifold meshes with similar shapes in general positions in space. This means that meshes represents the same or similar real world objects. Typically the same bone or muscle from different scanners and patients are expected to be used.

Every mesh can have any translation and rotation in 3D space. It can also be freely non-uniformly scaled compared to the others. There are also expected to be small deformations that are results of both differences in original scanning subjects and methods of resulting data processing.

The topology of mesh is specified to contain only triangular polygons. There is no assumption on quality of such triangles however it is not subject of the algorithm to perform any deliberate enhancement in this field and therefore degenerated triangles in the input are likely to be present in the output and might as well cause numerical problems during the transformation as well.

The mesh is not expected to be fully closed and may contain some smaller holes. It is however important, that those holes are small enough so that they can be filled using an automatic approach and most importantly, they do not split the mesh into two closed surfaced objects. If this happened, smaller component would

be trimmed out.

It is allowed for the meshes to contain irregularities such as non-manifold edges. Those edges can be located in one or all inputs. It is however important for the method, that for each surface element of the represented object, at least one input mesh contain manifold edges only. Otherwise the results of the method might not be fully correct. The approach also expects mesh surface not to intersect itself in any place. Such artifact is not detected by topology inspection and therefore not removed. It would later cause problems when surface normals are checked during registration.

The mesh must be genus 0 object. Without mathematical terms, this means that there is no tunnel in the mesh, such as e.g. in torus, and that mesh can be deformed to shape of sphere without any cuts. The holes that are not part of the original object but results of poor triangulation are, however, allowed.

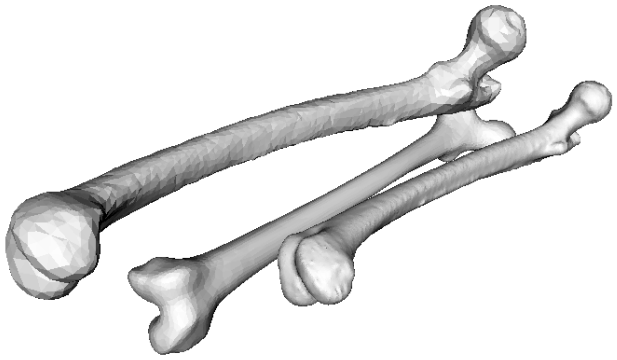


Figure 4.3: Sample of input. Three various femur bone surface meshes of similar outline shape but different topology with general positions in space.

4.2 Mesh topology refinement

Most of later parts of the algorithm expect input meshes to be manifold, closed and graph of each consisted of a single component. This is in contradiction with original assumptions. This step refines meshes so that they fulfil new requirements. It might however be complicated to reconstruct damaged parts of models without errors or loss of details. We will therefore try to recover missing information from corresponding parts of other meshes during the morphing step later.

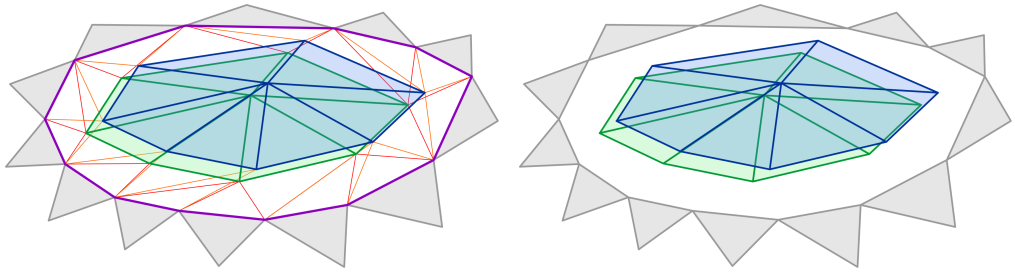
The fix of mesh consists of several consequent steps (see algorithm C.1). First, non-manifold edges are detected and removed. Then non-manifold triangles re-

maining in the mesh are found and removed. It might happen that some, hopefully small parts of mesh, become split from main component. These parts must be cut off. The last step is retriangularisation of holes in the mesh.

4.2.1 Non-manifold edges

Non-manifold edges are easy to detect just by counting triangles adjacent to each edge. Manifold edges in a manifold mesh have always one or two adjacent triangles. The case of a single triangle belongs to boundary of the hole in the mesh. In case of more than two triangles, the edge must be non-manifold.

In ideal case, two of the adjacent triangles could be called valid and the others removed. It might however be impossible to distinguish them in general case. For example if part of the surface was doubled and lying right on each other (see Figure 4.4a), the outline where the surface splits would consist of non-manifold edges with three adjacent triangles each. From topological aspect, both nearly overlapping neighbours would be equivalent.



(a) Original (b) After non-manifold removal

Figure 4.4: Doubled surface configuration (blue and green) connected with surrounding mesh (grey) by triangles (orange and red) with non-manifold edges (purple).

Therefore I decide to remove all those triangles instead rather than to risk wrong choice of left out triangles that might lead to manifold but invalid mesh surface.

This way all triangles around doubled surface area are removed splitting both layers away from the main mesh by circular hole (see Figure 4.4b). This problem will be solved later.

4.2.2 Non-manifold triangle fans

Even if all edges of mesh have two or less adjacent triangles, the mesh itself still does not have to be manifold. As can be seen in Figure 4.5, there might still

exists triangles that are connected to largest mesh graph component but do not share an edge with it.

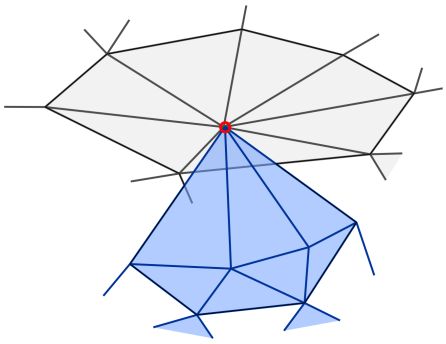


Figure 4.5: Non-manifold vertex (red) connecting two triangle fans (grey and blue).

In manifold mesh with closed surface, there is always exactly one closed fan of triangles. If there is a hole in the mesh, the fan will not be closed. If there are more than one holes sharing single vertex, there will be more than one fan of triangles around that particular vertex.

The situation is similar as in previous Section, just with edges replaced by vertices. Once again, I am not able to tell which fan consists of valid surface triangles and which consists of invalid inner triangles that must be removed.

I could therefore again test each vertex and find its triangle fans by going around the vertex. If there was only one fan, the vertex would be considered good. If there were more fans, I would remove all of their triangles, effectively removing the vertex from mesh.

This might however not be necessary in most cases. If there is at least one closed triangle fan around the vertex, I can say that it consists of valid triangles and leave it out from the removal process. In theory, there could be another closed fan at the same vertex and I might pick the wrong one. However it is both unlikely and safe to ignore, as the wrong fan that I left in the mesh would be isolated from the main mesh graph component as a result of the removal of all other triangles and removed later. This is guaranteed by previous step that removed all non-manifold edges.

I can therefore sort triangles around vertices by their adjacency, check if some of the detected fans are closed and if there is such, I can leave its triangles intact. This comes from assumption, that closed fans are more likely to be consistent with rest of mesh surface like in Figure 4.5. This way the vertex count of mesh is better preserved.

In some cases, like in Figure 4.6, more than one triangle fan might actually be

valid part of surface. Both fans are not closed and therefore removed. This decreases the quality of mesh, but it does not violate with later steps as it just merges two or more touching holes that already exists in the mesh.

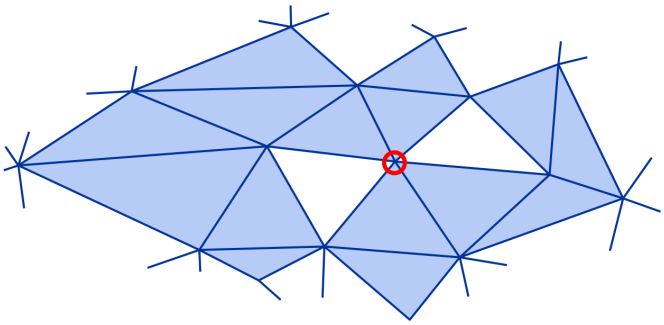


Figure 4.6: Two holes sharing vertex (red) causing false detection of non-manifold vertex based on triangle fan count.

4.2.3 Isolated mesh parts

After removal of non-manifolds, some parts of the mesh can become isolated from the main body. It might sometimes be obvious for observer how these parts should be connected together. However in most cases, I expect that these fragments will be just small clusters of triangles, mostly remainders of former non-manifold artifacts like in Figure 4.5.

Therefore I do not try to recover such isolated parts and remove them from the mesh data structure completely. I do it by *DFS* algorithm detecting mesh graph components and its sizes in number of vertices. I take the largest component as the valid mesh body and remove all vertices and triangles of the others. Assumption is that the vertex count of main component will be significantly larger than of others and minimum valid surface parts will be removed by the step.

I have also found out, that some isolated parts, such as single non-connected vertex, might be in the original input data. This might cause problems to some algorithms used later on, such as parametrisation. This step solves this problem automatically.

4.2.4 Holes

All previous refinement steps just removed vertices and triangles from the mesh. It is therefore likely to find holes in the output mesh surface at this moment. Parametrisation techniques usually expect the mesh to be closed. Luckily the

mesh is manifold and composed of single component now. Therefore it should not be a problem to find outline of each hole and re-triangularize it.

My first idea was to detect vertices with non-closed triangle fan and then traverse around the hole by triangles adjacent to edge. This way closed polyline would be found outlying the hole. It is even more simplified by the non-manifold triangle fan removal step before, as there can now be no two holes sharing same vertex (see Figure 4.6). The isolated component removal process then ensured that there will be no isolated island inside the hole, like in Figure 4.4b.

Therefore it would be sufficient then to split the general polygon to convex parts and triangularize it in simplest way of triangle fan from any point. Then series of edge flips could improve the local mesh quality by implying Delaunay condition of empty circumcircle.

However similar approach is already implemented in *VTK*. Therefore I used the implementation described in B.1.2 instead.

It might be also possible to improve quality of final mesh if large triangles were further tessellated to maintain original mesh resolution. It is unnecessary for most meshes, but it will improve output details if applied on the mesh, that is used as supermesh in morphing process. Such methods are described in papers [3] and [19]. An implementation in *vtkMEDFillingHole* class compatible with *VTK* is also available. These methods follow tangent of surface around the hole to produce smooth patches and even tessellate the space to maintain resolution of the original mesh. This could significantly reduce the distortion of filled regions but it was not tested for time reasons.

In this step we have inserted new triangles. However these triangles might not be valid. Once again, take an example in Figure 4.4b and imagine that the cut off central part was a high peak. Then the filled hole will be flat area not respecting original shape.

Then I expect that this peak will be preserved in other input meshes and will provide necessary data during morphing. As we do not want to influence that process by newly inserted unreliable triangles, I assign them reliability weights. I used simple weight metric when all original triangles have weight of 1.0 and all newly inserted have weight of 0.0. This way the local morphing will use only data from other meshes if local triangle is unreliable.

4.3 Initial registration

It is important to point out that distinguished features, e.g., the ball of femur bone, of one mesh might, in space, lie quite far from the same feature of another mesh since models are freely positioned.

We, therefore, have to perform global registration first to roughly align the meshes such that similar parts are positioned close to each other. For this purpose, we use *principal component analysis* (*PCA*) [12] (see alg. C.2) as described in this section.

4.3.1 Principal Component Analysis

It is important not to expect that similar parts of meshes are the nearest one in the input as models are positioned in general way. We therefore have to perform global registration first before finer alignment can be achieved locally in next steps.

PCA transforms coordinate system in such way, that the most important direction is in the first component [12]. It does it by inspecting correlations in data.

First the covariance matrix is built using only space coordinates of mesh vertices. This matrix consists of auto-correlations of individual vertices $\overrightarrow{x}$ from mesh X with N vertices [12]:

$$M_C = \frac{1}{N} \sum_{\overrightarrow{x_i} \in X} (\overrightarrow{x_i} - \overrightarrow{\overline{x}}) \cdot (\overrightarrow{x_i} - \overrightarrow{\overline{x}})^T \tag{4.1}$$

where $\overrightarrow{\overline{x}}$ is the centre of gravity of the mesh gained as:

$$\overrightarrow{\overline{x}} = \frac{1}{N} \sum_{\overrightarrow{x_i} \in X} \overrightarrow{x_i} \tag{4.2}$$

Then the eigenvalues and eigenvectors of M_c are found. The eigenvector for largest eigenvalue then contains direction of main object axis which is the dominant direction in the object. The Figure 4.7 presents this output on femur bone model.

The longest axis line along the bone is the main direction of the bone. The other two directions are found in remaining two eigenvectors and are orthogonal to each other as well as the first axis.

The relative direction in respect to mesh orientation as well as the order of axis is maintained over different representations if meshes are reasonably similar in outline, have similar vertex distribution - not necessarily density and therefore vertex count, and have features distinguished enough so there is safe difference between individual eigenvalue sizes.

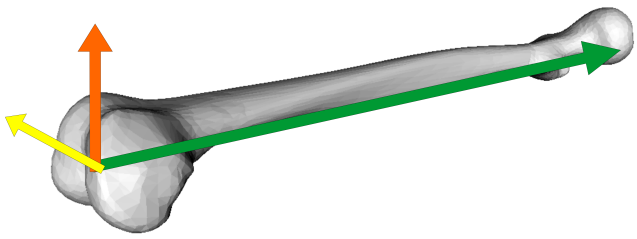


Figure 4.7: Main object axes for a single input mesh. Green arrow for axis of the largest eigenvalue, orange for the middle and yellow for the smallest.

An example of meshes where this is not preserved is close to sphere ellipsoid. Such mesh would have very similar eigenvalues and minor differences in distribution of vertices on surface could cause change in principal axis order.

Experiments with sample data shows that method is safe to use with our meshes.

Now we have basis of coordinate systems for all input meshes and we can use them to get proper transformation that would ensure proper alignment (see algorithm C.8).

First we have to enrich the three axes by origin to have complete coordinate system. Simple centre calculated as mesh vertices average can be used with reasonable vertex distribution (close enough to uniform). Centre of object oriented bounding box can provide better results otherwise.

The initial rigid body transformation is then combination of translation to origin of coordinate system and rotation to unify all main axes. One model can be chosen to define target position if maintaining the position is desired or zero point and default x, y, z axes can be used otherwise.

Translation of the mesh i to the origin point $(0, 0, 0)$ is specified simply by a difference of coordinate system origin points so that the translation matrix $\boldsymbol{T}$ is build using

$$\boldsymbol{T}(i, \mathbf{0}) = \begin{bmatrix} 1 & 0 & 0 & -C_{ix} \\ 0 & 1 & 0 & -C_{iy} \\ 0 & 0 & 1 & -C_{iz} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.3}$$

where $\overrightarrow{C_i}$ is centroid of i-th mesh serving as local coordinate system origin point.

Rotation matrix is more complex as it must consist of several partial rotations. Firstly the mesh i is rotated so that its axis x aligns with base axis $(1, 0, 0)$.

Rotation axis is vector perpendicular to both source and target vectors of x axis:

$$\overrightarrow{o} = \frac{\overrightarrow{axis_x} \times (1,0,0)}{|\overrightarrow{axis_x} \times (1,0,0)|} \tag{4.4}$$

Then the rotation angle $\alpha_{\overrightarrow{x}\to\overrightarrow{0}}$ is calculated from its sin and cosine values, obtained as sizes of cross product and dot product:

$$\alpha_{\overrightarrow{x}\to\overrightarrow{0}} = \arctan \frac{|\overrightarrow{x} \times \overrightarrow{0}|}{|\overrightarrow{x} \cdot \overrightarrow{0}|} \tag{4.5}$$

A rotation matrix for general rotation around normalised axis o can be build as [8]:

$$\boldsymbol{R}(\alpha_{\overrightarrow{x_i}\to\overrightarrow{0}}, \overrightarrow{o}) = \begin{bmatrix} \overrightarrow{o_x}^2 \cdot (1-c) \cdot +c & \overrightarrow{o_x} \cdot \overrightarrow{o_y} \cdot (1-c) - \overrightarrow{o_z} \cdot s & \overrightarrow{o_x} \cdot \overrightarrow{o_z} \cdot (1-c) + \overrightarrow{o_y} \cdot s & 0 \\ \overrightarrow{o_y} \cdot \overrightarrow{o_x} \cdot (1-c) + \overrightarrow{o_z} \cdot s & \overrightarrow{o_y}^2 \cdot (1-c) + c & \overrightarrow{o_y} \cdot \overrightarrow{o_z} \cdot (1-c) - \overrightarrow{o_x} \cdot s & 0 \\ \overrightarrow{o_x} \cdot \overrightarrow{o_z} \cdot (1-c) - \overrightarrow{o_y} \cdot s & \overrightarrow{o_y} \cdot \overrightarrow{o_z} \cdot (1-c) + \overrightarrow{o_x} \cdot s & \overrightarrow{o_z}^2 \cdot (1-c) + c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.6}$$

where $c = \cos(\alpha_{\overrightarrow{x}\to\overrightarrow{0}})$ and $s = \sin(\alpha_{\overrightarrow{x}\to\overrightarrow{0}})$.

Similar approach is then used for rotation matrix $\boldsymbol{R}(\alpha_{\overrightarrow{y_i}\to\overrightarrow{0}})$ for alignment of resulting axis y to basic $(0,1,0)$ and then in reverse order for matrices rotating basic axis vectors to match y and x axis of mesh j.

The total rotation is then gained by multiplication

$$\boldsymbol{R}(i,j) = \boldsymbol{R}(i,\boldsymbol{0}) \cdot \boldsymbol{R}(\boldsymbol{0},j) = \boldsymbol{R}(\alpha_{\overrightarrow{x_i}\to\overrightarrow{0}}) \cdot \boldsymbol{R}(\alpha_{\overrightarrow{y_i}\to\overrightarrow{0}}) \cdot \boldsymbol{R}(\alpha_{\overrightarrow{0}\to\overrightarrow{y_j}}) \cdot \boldsymbol{R}(\alpha_{\overrightarrow{0}\to\overrightarrow{x_j}}) \tag{4.7}$$

I also add non-rigid transformation at this point to adjust model scaling. Simple scaling matrix is built using oriented bounding box sizes of meshes i and j. Orientation of those boxes is also determined by main axes calculated above. Therefore scaling matrix $\boldsymbol{S}(i,j)$ adjusting size of mesh i to size of mesh j is

$$\boldsymbol{S}(i,j) = \begin{bmatrix} \frac{size_{xj}}{size_{xi}} & 0 & 0 & 0 \\ 0 & \frac{size_{yj}}{size_{yi}} & 0 & 0 \\ 0 & 0 & \frac{size_{zj}}{size_{zi}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.8}$$

Then final transformation matrix $\boldsymbol{M}(i,j)$ aligning mesh i to mesh j can be gained from combination of translation, rotation from mesh i to basic position, scaling and transformations from basic position to position of mesh j in reversed order:

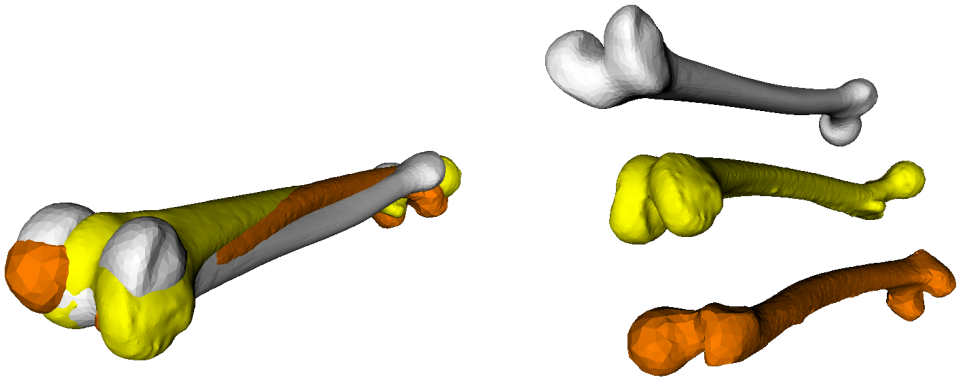
$$\boldsymbol{M}(i,j) = \boldsymbol{T}(i,\mathbf{0}) \cdot \boldsymbol{R}(i,\mathbf{0}) \cdot \boldsymbol{S}(i,j) \cdot \boldsymbol{R}^{-1}(j,\mathbf{0}) \cdot \boldsymbol{T}^{-1}(j,\mathbf{0}) \tag{4.9}$$

Therefore each vertex of each input mesh is transformed by only single vertex - matrix multiplication.

In theory, all meshes would be approximately aligned at this moment. The quality of alignment is limited by non-rigid deformations and differences of vertex distributions in meshes influencing directions of the main axes.

4.3.2 Final alignment

In reality, the direction of main axes is not guaranteed to have always the same sign. Therefore the final direction for each axis has to be determined other way. If this was not done, results like in Figure 4.8 would be likely to occur.



(a) Aligned (b) Moved apart

Figure 4.8: Incorrect PCA alignment of three femur bones if orientations of main axes are not checked.

The easiest way to fix this is using distance between pair of meshes as error measure. Then only six possible positions given by six combinations of two flips of two axes x and z must be tested. The third axis z is fixed by the previous two to keep transformation rigid.

Therefore for alignment of mesh i to j is finalised by testing initial and three additional transformations rotating mesh i by 180° around x, y and both axis. The transformation with smallest error measure is then picked as final alignment.

Easiest and fastest way of error distance calculation is using sum of square distances between nearest vertices. First problem is that this metric is not symmetric. It means that if input order changes, different distance is gained between the same pair of meshes.

The difference growths when low polygonal mesh A lies near rich mesh B. Then the distance from A to B is small as there is always some near vertex from B to pair with for any vertex of A. This, however, does not apply to the other order and, therefore, the result is significantly different even if the final sum is normalised by number of measured distances.

It might still output same alignment but for consistency reasons I find nearest distances from mesh A to B and then from B to A so I can use their average as final distance. This guarantees symmetric results.

Additionally I also avoid the above mentioned mesh size difference problem complexly by replacing vertex to vertex distances by vertex to nearest triangle distance (see algorithm C.9).

Given two meshes X_i and X_j, distance between each vertex $\overrightarrow{x_i}$ of mesh X_i and every triangle T_j of mesh X_j is tested yo find the nearest one.

First, plane ρ_j for triangle T_j of mesh X_j is found. Point $\overrightarrow{x_i}^\rho$ is then defined as projection of vertex $\overrightarrow{x_i}$ to the plane ρ.

If $\overrightarrow{x_i}^\rho$ lie in T_j then its square distance to $\overrightarrow{x_i}$ is distance from T_j. Otherwise square distance to nearest T_j edge is used. This is subject of minimisation over all triangles of T_j:

$$dist^2(\overrightarrow{x_i}, X_j) = \min_{T_j \in X_j} dist^2(\overrightarrow{x_i}, T_j) \tag{4.10}$$

(4.11)

$$dist^2(\overrightarrow{x_i}, T_j) = \begin{cases} |\overrightarrow{x_i} - \overrightarrow{x_i}^\rho|^2 & \text{if} \overrightarrow{x_i}^\rho \text{lies in} T_j \\ \min\limits_{edge \in T_j} dist^2(\overrightarrow{x_i}, edge) & \text{otherwise} \end{cases}$$

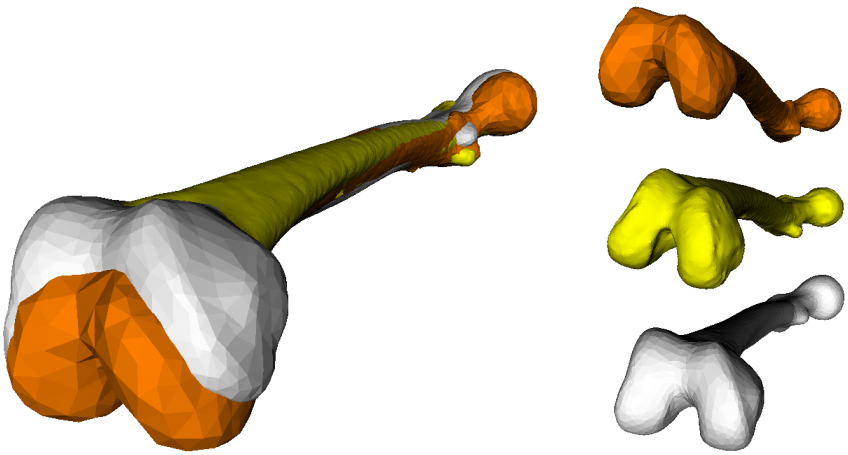
Formula for $dist^2(\overrightarrow{x_i}, edge)$ is similar.

Then both meshes are swapped and second set of distances is calculated and combined weighted by size of individual meshes.

$$e(X_i, X_j) = \frac{1}{2|X_i|} \sum_{\overrightarrow{x_i} \in X_i} dist^2(\overrightarrow{x_i}, X_j) + \frac{1}{2|X_j|} \sum_{\overrightarrow{x_j} \in X_j} dist^2(\overrightarrow{x_j}, X_i) \tag{4.12}$$

Transformation with minimal error distance $e(X_i, X_j)$ is chosen as final transformation aligning mesh i to j.

This way every i^{th} input mesh is aligned with selected mesh j. Mesh j can be any of input meshes as it only affects position of output and does not influence output quality. The result is displayed in Figure 4.9.



(a) Aligned (b) Moved apart

Figure 4.9: Successful rough PCA alignment of three femur bones.

Complexity is linear in number of input meshes as each of them is aligned only once. The calculation of main axes is linear in vertex size of mesh. Same applies to calculations of centres and bounding boxes.

Only part that can achieve quadratic complexity is distance error measure in last step. This is when each vertex to each triangle is tested. Binary space partitioning structures can be used to improve this behaviour. Uniform grid, kd-tree or bd-tree can be used to find nearest triangle candidates. This reduces number of candidates for distance test.

It is also unnecessary to measure distance of models using the original meshes. Coarse meshes gained by edge decimation or progressive hulls can be used instead. I used filter described in B.1.3 to produce meshes of few hundred vertices. This improved the performance significantly (see comparison in Chapter 5.3.1).

4.4 Non-rigid ICP registration

Now we have all input meshes in condition suitable for a more sophisticated surface analysis. We could now proceed with parametrisation, but as comparison of different alignment techniques in Section 5.3.3 shows, the product of *PCA*

is far from ideal. Although this should cause no problem in morphing theory as different objects should be possible to morph, it is necessary to synchronise parametrisations to have feature points on same places. Our experiments described later show that it is necessary to align all meshes even more precisely before parametrisation.

I therefore use base of algorithm from article [5] for slightly non-rigid *ICP* registration described in original form in Section 2.2.1. I just modified point neighbourhood selection part and final deformation approximation. The review of the method is provided in this section.

The registration process is binary while they always only work with one source mesh and one target mesh. For this reason, I will describe the rest of algorithm on two meshes only. At the end, only one target mesh will be chosen and all other meshes will be registered as source meshes to align with this target. Those registrations are run separately.

Choice of target mesh in this context is not very important as we only want meshes to have the same shape. However I prefer the largest mesh in number of vertices to be target as this will allow to find better candidates for source mesh's closest points int the *ICP* part. It is also faster to deform smaller source meshes. The sampling of feature points on small meshes can be faster as well, depending on the method used.

Principle of algorithm that registers one source mesh to one target mesh is as follows (see algorithm C.3 for more details):

1. Pick one target mesh and one source mesh
2. Pick feature points in the source mesh (Section 4.4.1)
3. Pick region for each feature point (Section 4.4.1)
4. Register each region in target mesh using *ICP* (Section 4.4.2)
5. Interpolate all registrations to get final transformations per source mesh vertex (Section 4.4.3)
6. Transform source mesh vertices accordingly to get non-rigid aligned source mesh with target mesh (Section 4.4.3)

Now, the detail description will be provided.

4.4.1 Single ICP region selection

We have already picked one source and one target mesh. Let us consider source mesh X_i and target mesh Y. We first pick some number of feature points from X_i.

I use the same approach as [5] so I pick points randomly. I will want to achieve good local registrations and I therefore pick 10% of mesh vertices, but at most 500 in absolute count. That prevents redundancy of identical transformations for very close feature points in large meshes.

For each such feature point, *ICP* will be run as will be described in the next section and therefore some other points that describe its neighbourhood must be picked. Original article [5] suggest stochastic selection of points based on their distance and normal direction variety. This selects mostly the vertices in the neighbourhood, but also adds vertices from other side of the object to describe the global shape and avoid problems in flat area's of the mesh. See Figure 4.10.

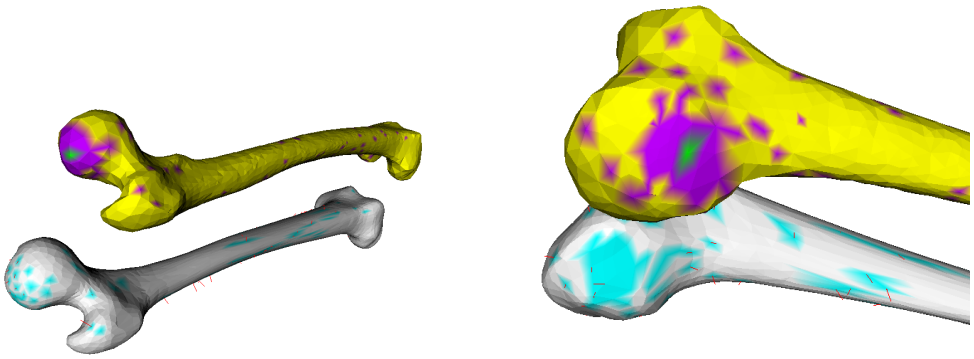


Figure 4.10: Source mesh (yellow) with single selected feature point (green) and its region points (purple) registered to nearest points (cyan) of target mesh (white). Region points sampled using method from [5].

However, during experiments I have found out, that such selection is not local enough to provide good local alignment (see Section 5.3.2).

Therefore I modified the selection to pick points closer to the feature point. We already have model reasonably well aligned using *PCA*. If this was not true, then rigid body *ICP* could be run as described in 2.1.1.

This way two matching surfaces are very near to each other and we can assume that we do not need that much information about global position to perform successful registration.

This way I always pick only 3-neighbourhood of the feature point to be the region for the *ICP* (algorithm C.10). See Figure 4.11 for comparison.

As comparison in 5.3.2 shows, good alignment of surfaces can be found with this selection.

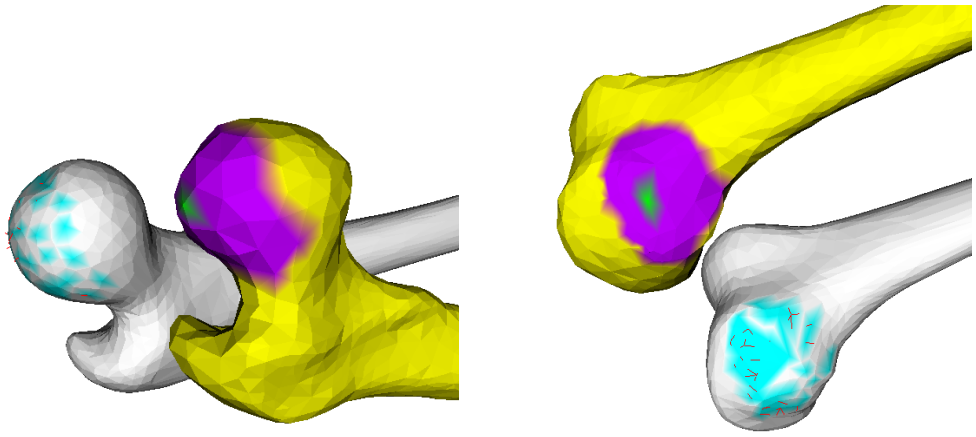


Figure 4.11: Source mesh (yellow) with single selected feature point (green) and its region points (purple) registered to nearest points (cyan) of target mesh (white). Region points sampled using 3-neighbourhood.

4.4.2 Iterative Closest Point

Now each region built in the previous section is registered with the target mesh. I use the original *ICP* description from [4] here. It is well described in theoretical Chapter 2.1.1. Schema is also presented in algorithm C.11.

Change to the original *ICP* by Besl [4] was made in the nearest point to point search mechanism. I have added normal check to ensure, that both aligned surfaces have similar normal orientation. I do not allow nearest point to have normal with angle difference to checked point's normal larger than 90°. Therefore condition is stated such as:

$$\overrightarrow{n_i} \cdot \overrightarrow{n_j} \geq 0$$

The normals are estimated as average of point triangle normals weighted by their inner angle at point:

$$\overrightarrow{n_i} = \frac{\sum_{T \in cellsof \overrightarrow{p_i}} |\angle \overrightarrow{a}\, \overrightarrow{p_i}\, \overrightarrow{b}\,| \cdot (\overrightarrow{a} - \overrightarrow{p_i}) \times (\overrightarrow{b} - \overrightarrow{p_i})}{\sum_{T \in cellsof \overrightarrow{p_i}} |\angle \overrightarrow{a}\, \overrightarrow{p_i}\, \overrightarrow{b}\,|}$$

$\overrightarrow{a}$, $\overrightarrow{p_i}$, $\overrightarrow{b}$ are vertices of triangle T.

This is applied in each iteration of error minimisation algorithm described in 2.1.1.

The output is a final transformation matrix M that is gained from each iteration of ICP as

$$\begin{aligned} M_0 &= \boldsymbol{I} \\ M_{i+1} &= T_{i+1} \times M_i \\ M &= M_{max} \end{aligned}$$

where T is transformation gained in ICP iteration as combination of rotation and translation in this order. See Chapter 2.1.1 for more details again.

As not all points but only earlier selected neighbourhood points are used for the source mesh, only the alignment errors in the feature point's neighbourhood are fixed. This might cause other parts of the mesh to be miss-aligned (see 4.12). That is however no problem as non-rigid deformation of the mesh surface will handle it later.

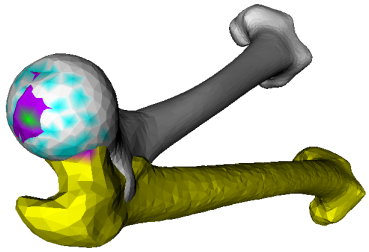


Figure 4.12: Final transformation of source mesh (yellow) to target mesh (white) according to local ICP result for region (purple) of single feature point (green) on bone's head. Region sampled using 3-neighbourhood.

4.4.3 Deformation of source model

After previous step, each pair of source and target mesh has K feature points with k transformation matrices M_j that describe alignment deformation for their neighbourhood. Now we must expand this information to whole source mesh surface and therefore completely non-uniformly transform source mesh vertices to achieve alignment with the target mesh. As individual vertices can be transformed in a different way, this leads to non-rigid deformation of mesh. The result

will, therefore, be a slightly deformed source mesh that minimises the point-to-point distance to the target mesh.

First the feature point references are filtered, then their transformations are interpolated into the rest of mesh and finally the individual vertices are transformed.

Some feature points may be inconsistent with rest of pairs. This is when distance between two feature points and their transformed image differ too much. This distance should be calculated as geodesic to avoid miss judgements of bended areas. In our case such problems are not likely to happen, because we had two nearly complete and very similar objects at the beginning. The authors of [5] worked with incomplete scan samples instead.

Regardless if we filtered some pairs out, we now have a set of pairs usable to deform source mesh X.

Article [5] uses thin-plate splines to distribute transformations across complete surface (more about this in 2.2.1). I experimented with simpler method using non-linear distance based interpolation.

I begin with k feature points of source mesh X with K transformation matrices M_j. Then each vertex of mesh X is evaluated by distance from each of feature points.

Regardless of measure chosen, the next step weights influence of all feature points $\overrightarrow{x_j}$ based on distance from inspected general vertex $\overrightarrow{x_i}$:

$$w_{i,j} = \begin{cases} 1 - \left(\frac{1}{d_{MAX}} \cdot distance(\overrightarrow{x_i}, \overrightarrow{x_j})\right)^c & \text{if } distance(\overrightarrow{x_i}, \overrightarrow{x_j}) < d_{MAX} \\ 0 & \text{if } distance(\overrightarrow{x_i}, \overrightarrow{x_j}) \geq d_{MAX} \end{cases}$$

where d_{MAX} is maximum distance for which the feature point can influence vertex. c is power of distribution influence decrease. I have chosen d_{MAX} to be 0.1 an $c = 1.5$ in my experiments but I tested only few other values and kept the best of them. It allows distribution of most of deformation to closest neighbourhood.

The function $distance(\overrightarrow{x}, \overrightarrow{y})$ can be either geodesic or Euclid distance. However in both cases must be reasonably normalised so that the maximum measured distance is 1. This enables d_{MAX} to remain constant for any scale model. Good approximation or normaliser for Euclid measure is size of oriented bounding box. For geodesic distance function we usually have the maximum from pre-calculated values.

Sum of influences of all feature points cannot be bigger than one. At the same time, no vertex can be left out without any deformation. That would cause such points to be dragged out of the mesh. Therefore I use conditional normalising to fix sums that are not zero and I distribute all feature point wights equally otherwise:

$$w_{i,j}^{N} = \begin{cases} \frac{w_{i,j}}{\sum_i w_{i,j}} & \text{if } \sum_i w_{i,j} > 0 \\ \frac{1}{k} & \text{if } \sum_i w_{i,j} = 0 \end{cases}$$

Thanks to smoothness of decay of weight function, all such vertices would be possibly miss-aligned but will not create any sharp edges.

These weight are then used to interpolate between deformation matrices for each of K feature point giving final position of inspected source mesh vertex $\overrightarrow{x_i}$:

$$\overrightarrow{x_i'} = \sum_{j=0}^{K} \left(w_{i,j}^{N} \cdot M_j \right) \cdot \overrightarrow{x_i} \tag{4.13}$$

The process is repeated with each vertex $\overrightarrow{x_i}$ of source mesh. Final output is source mesh non-rigidly transformed thus deformed to better align with target mesh (see Figure 4.13).

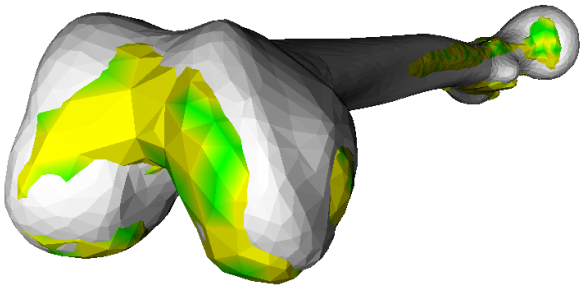


Figure 4.13: Output of non-rigid ICP alignment step for two femur meshes. The source mesh is yellow, the target mesh is white. Green spots marks feature points.

The distance used in the interpolation should be measured as geodesic distance which could be approximated using edge-length weighted distance in graph found using *Dijsktra* or *Floyd–Warshall* algorithm for shortest paths in graph.

However practical tests approved usage of simple Euclid distance as well if the initial alignment is good and feature point density is high enough. Usage of Euclid measure has benefit of fast $O(N)$ time complexity without need of external storage. Disadvantage is hidden in situations where two parts of mesh are facing each other in close proximity while their true geodesic distance is high (e.g. nearly touching hands).

Evaluation of alignment precision and visualisation of outputs can be found in Section 5.3.3.

4.5 Spherical parametrisation

Now all meshes have approximately same shape. In this phase, we will split the two variants of the main method. This and the following section are only relevant to the spherical domain version as was announced in the very beginning of Chapter 4. For this, we have to create suitable parametrisation of all meshes. These parametrisations will help us find bijective relations between meshes in the later morphing phase described in Section 4.6.

4.5.1 Basics

In theory, every manifold genus 0 mesh can be parametrised on unit sphere. We can imagine such process as expansion of inner volume of model similar to pumping air into balloon.

In practise however, the process is simple and accurate only for "star shaped" objects. Such an object has an inner point from which every part of surface can be seen without occlusion. Then given such point, mesh just needs to be centred to it and all vertex coordinates normalised to achieve valid parametrisation. We call parametrisation valid if it maintains topology with original mesh, therefore their graphs are isomorphic, and no two triangles are overlapping (see Figure 3.2 for example of artefact).

Overlap can be easily measured by inspecting numbers of clockwise and counter-clockwise oriented triangles. Parametrisation is valid if all triangles are oriented in the same way when observed from sphere centre.

Such test is easily implemented using simplified volume formula such as in article [1]:

$$\tilde{V}(T_{a,b,c}) = (\overrightarrow{a} \times \overrightarrow{b}) \cdot \overrightarrow{c} \tag{4.14}$$

If our triangles are oriented clockwise then parametrisation error of parametrisation $P(X)$ of mesh X can be stated as sum of negative test results:

$$error(P(X)) = \sum_{T \in X} \begin{cases} |\tilde{V}(T_{a,b,c})| & \text{if } \tilde{V}(T_{a,b,c}) < 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.15}$$

This does weight each triangle by its area so that the final error depends not only on number but also on size of overlapping triangles.

In ideal situation, error measure of 0 should be achieved. Practise shows that it may be hard for some meshes and that errors of size 10^{-6} and below can be ignored. However even if an error is exactly zero it does not say anything about quality of triangles in parametrisation.

Ideal parametrisation have triangles of similar sizes. This is not possible to achieve for complicated objects like cow model with head and legs. It is however important for the result that the smallest triangles do not degenerate to almost zero area, which would cause barycentric coordinates to be heavily influenced by numerical errors. As far as I know, there is no method that solves this and all simply relies that it will end up good enough automatically.

4.5.2 Projection relaxation

As we do not work with strictly "star shaped" meshes, we have to deal with overlaps after initial projections. I have tested three different approaches to this problems and none of them seems to do it very well.

The First method was Alexa's relaxation algorithm based on Gauss-Seidel iterative solution of linear equations. It is originally described in [1] and is more explained here in Section 3.2. I have tested both implementation of Ing. Parus Ph.D. and my own to minimise risk of implementation mistake or wrong interpretation of original article.

The second method is very similar to the previous, yet simpler and more recent. It was published in [27] and discussed in 3.5. The approach is almost identical as only difference in relaxation schema is in equation for next-step parametrisation coordinates. This means that I only had to replace equation 3.3 by 3.6.

The third method described in [2] was result of my search on internet for reference implementations. For testing purpose, I have used author's implementation of method. This method is again based on iterative solution of linear equations. The main feature of the implementation is the ability to use *GPU* through *OpenCL*. Therefore it can hardly be compared with my *CPU* implementations by time.

The results of comparison of these methods can be found in Section 5.4.1. The main outcome is that not a single method was able to produce valid parametrisation of all tested input meshes. Therefore I decided to further work with Alexa's method which is easy to implement as I need to be able to modify it in later parts.

First problem with implementation is initial selection of the projection point. It is vital that it lies inside the mesh in the beginning. Otherwise, degenerative solu-

tions are found (see 4.14). Its choice can influence convergence of later relaxation process [20].

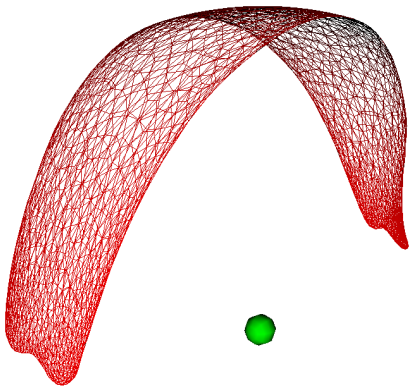


Figure 4.14: Spherical parametrisation with centre out of original volume. Projection centre point highlighted by green dot.

All methods, I have read about, either do not discuss the problem at all or they expect user interaction or they expect centre of gravity to suffice inside volume condition.

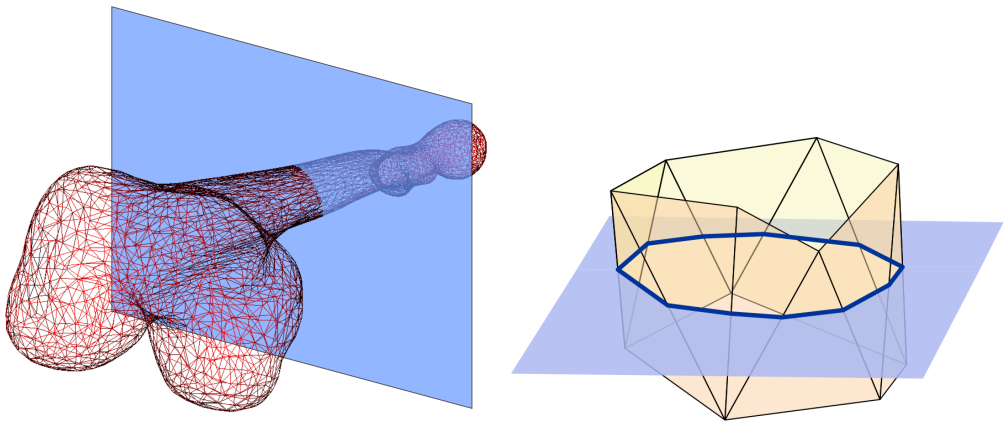
My experiments showed that none of these approaches can be used for most of models in automatic framework. Therefore I suggest a different technique for inner point selection. We want the inner point to be inside of mesh and to be position in such a place that the projection overlap error is minimised. I suggest usage of main axes discussed in Section 4.3.1 (see algorithm C.12) instead of implementation of complicated mathematical mechanism to express those conditions using equations listed above (see eq. 4.15)

From knowledge of my input mesh shapes, which are muscles and bones, I expect the middle of the mesh to be good enough. I just need to move the centre of gravity found by equation

$$c = \frac{1}{|X|} \sum_{\vec{x} \in X} \vec{x} \tag{4.16}$$

to the inner volume. Therefore I use plane defined by the centre of gravity and two minor axis from *PCA*. Then I find triangles that intersect this plane. This gives me polygon of intersection of mesh and constructed plane (see Figure 4.15).

Now the problem is reduced to 2D. I then calculate new centre of gravity for polygon points only. This one could still lie outside. Therefore I either use one of two main vectors in plain again or find new main vector by *PCA* in 2D space of



(a) Cut of mesh (b) Detail of cut

Figure 4.15: Intersection of plane specified by two minor axes from PCA together with gravity centre of mesh and mesh surface itself. Intersected triangles and their points form base for inner centre point selection.

point to plane projections and check number of intersections with polygon from the new centre point in one direction. If number of intersections is even number, point lies outside and is moved to the average of two consequent intersections.

It is important to notice that this approach is by far not general. It relies on meshes to be genus 0 and closed, which I can guarantee. In practise my inputs also provide reasonable shape so that initial position calculated from 2D cut always lies inside the mesh.

Gained point is then used as centre point for all experiments with parametrisation presented in this thesis.

In the Alexa's solution of the parametrisation problem[1], algorithm starts with initial projection of mesh to unit sphere. This is just normalisation of vertices moved to the centre $\overrightarrow{c}$ constructed at the end of previous Section:

$$\overrightarrow{x_i}^P = \frac{\overrightarrow{x_i} - \overrightarrow{c}}{|\overrightarrow{x_i} - \overrightarrow{c}|} \tag{4.17}$$

Then relaxation by iterative recalculation of each single parametrisation vertex $\overrightarrow{x_i}^P$ of mesh X is applied using formula

$$c = \frac{1}{\max|\overrightarrow{v} - \overrightarrow{x_i}^P|} \tag{4.18}$$

$$e(\overrightarrow{x_i}^P) = c \cdot \frac{1}{|N(\overrightarrow{x_i}^P)|} \sum_{\overrightarrow{v} \in N(\overrightarrow{x_i}^P)} \left((\overrightarrow{v} - \overrightarrow{x_i}^P) \cdot |\overrightarrow{v} - \overrightarrow{x_i}^P| \right) \tag{4.19}$$

$$\overrightarrow{\widetilde{x}_i}^P = \frac{\overrightarrow{x_i}^P - e(\overrightarrow{x_i}^P)}{|\overrightarrow{x_i}^P - e(\overrightarrow{x_i}^P)|} \tag{4.20}$$

where $N(\overrightarrow{x_i}^P)$ denotes neighbour vertices. See algorithm C.13 for overview.

The algorithm stops when the parametrisation error according to formula 4.15 becomes zero or when the maximum allowed iteration count is reached. I test the error only each 10th iteration to save time although the error calculation is about ten times faster than the iteration itself.

I use 1 000 or 10 000 as iteration maximum, but the changes after several hundred iterations tends to be minimum as can be seen in Section 5.4.1.

I also experimented with value of constant c in the formula 4.18, but it did not have much effect (see Section 5.4.1).

It is also possible to use intermediate results of relaxation as inputs of the running iteration. This is called *in-place* modification and it seems to improve not only the speed but also final parametrisation error (see Section 5.4.1). In this case, some vertices in the neighbourhood are results of previous and some of current iteration. This tends to be able to add more movement to the mesh fixing more overlaps in the process. Original text [1] does not specifically mention which variant to use. I decided for the in-place one.

4.5.3 Cascade schema

Regardless of any parameters, this method does not work for big meshes, like 42 502 vertex model of femur bone in Section 5.4.1. The reason seems to be, that there is not enough space for the vertices to be moved in the relaxation phase. I therefore came with an idea of cascade solution. This way the initial parametrisation is done on relatively small mesh with several hundreds vertices where triangles are big enough for relaxation mechanism to handle their movement as there is enough space in each vertex's neighbourhood to move in (see algorithm C.4). The initial parametrisation is then propagated up to the original mesh size.

I have used decimation filter described in Section B.1.3 to create coarse hull for the input mesh. I did not limit to single step only as this would cause large errors

in parametrisation after scaling up (see Section 5.4.3 for comparison). Therefore I choose factor two for mesh vertex count scaling.

The process starts with the nearest smaller power of scaling coefficient multiplied by base mesh size compared to vertex count of largest mesh. So I choose starting coarse mesh size as

$$N_{X_0} = N_B \cdot c^{\lfloor \log_c \frac{\max(N_{Xi})}{N_B} \rfloor} \tag{4.21}$$

where c is scaling coefficient, chosen to be 2, and N_B vertex count of smallest coarse mesh where the initial parametrisation is run, chosen to be 300 vertices.

So given the largest mesh to be femur bone with 42 502 vertices and other two meshes having 2 502 vertices only, initial coarse mesh will have 38 400 vertices. That would mean up-scaling for the smaller meshes. It would make no sense so the high-order steps are skipped for such meshes. On implementation level, their coarse meshes are identical to the input mesh if the target vertex count is higher than in input.

The algorithm cannot rely on meshes to have exactly same number of vertices anyway as the filter does not guarantee that the target vertex count will be obeyed absolutely.

The relation between larger and smaller mesh in the cascade is given by *mean value coordinates* (*MVC*) originally described in [24]. They allow expressing coordinates of point inside both convex and non-convex triangular mesh in similar manner as position of point inside triangle is determined by barycentric coordinates. They also work outside the mesh but as my previous application of them in bachelor thesis showed, the precision in outer volume is limited [13]. This is the reason why the progressive outer hull filter was developed by David Cholt [6] to create strictly outer hull of input mesh and therefore the *MVC* should be reliable for backward mesh reconstruction and propagation of parametrisation up to the original input sizes. Thanks to this, single vertex $\overrightarrow{x_i}$ of larger mesh X_k could be reconstructed from smaller mesh X_{k+1} using equation

$$\overrightarrow{x_{k,i}} = \sum_{j=0}^{N_{X_{k+1}}} w_{ij} \cdot \overrightarrow{x_{k+1,j}} \tag{4.22}$$

where w_ij is mean value coordinate of i-th vertex of bigger X_k in direction of j-th vertex of smaller mesh X_{k+1}.

This way cascade of coarse meshes is created so that the first meshes are the original inputs, then the coarse meshes with above mentioned exception for small inputs are created to approximately fulfil size given by equation 4.21.

Next meshes are consequently created to reach size $N_{X_{k+1}}$ reduced by coefficient c:

$$N_{X_{k+1}} = \frac{1}{c} \cdot N_{X_k}, k > 0 \tag{4.23}$$

Final coarse meshes will have basic size N_B. These final meshes are parametrised using algorithm of Alexa described in previous chapter. Then same mean value coordinates that maps vertex $\overrightarrow{x_i}$ of larger mesh X_{k+1} to linear combination of vertices of smaller mesh X_k are used to propagate parametrisation to higher mesh in similar way as in equation 4.22. Only the coordinates of points x_i are replaced by coordinates of parametrisations:

$$\overrightarrow{x_{k,i}}P = \sum_{j=0}^{N_{X_{k+1}}} w_{ij} \cdot \overrightarrow{x_{k+1,j}}P \tag{4.24}$$

After any parametrisation is projected to the bigger mesh, it is, likely however, no longer to be valid parametrisation. First of all individual parametrisation coordinates do not have to necessarily lie on unit sphere. This is because the larger mesh can have more features than the small one. Such features can then cause movement of parametrisation from or to the projection centre. This is easily fixed by normalisation of recalculated parametrisation vertices. Next, some of large model features that are omitted on small model can even cause some triangles to overlap on parametric domain. An example can be seen in Figure 4.16. For this reason new process of Alexa's relaxation is run to fix local problems.

Backward parametrisation projection and relaxation then repeats until original input size of mesh is reached. Based on this description, these steps can be done with individual meshes separately. However following Section will show that there are relations among parametrisations that imply simultaneous calculations and mutual adjustments.

4.5.4 Registration of parametrisations

Now we have a spherical parametrisation for each of the input meshes. Therefore we could proceed with morphing using barycentric coordinates as described in Chapter 3.1. However if we did, the results would be rather unpleasant. The Figure 4.17 demonstrates the problem. It shows result of morphing of two different models of femur bone with same vertex size.

The reason is as follows. As the parametrisation of each mesh is an independent process, there is nothing that would cause same parts of the described model

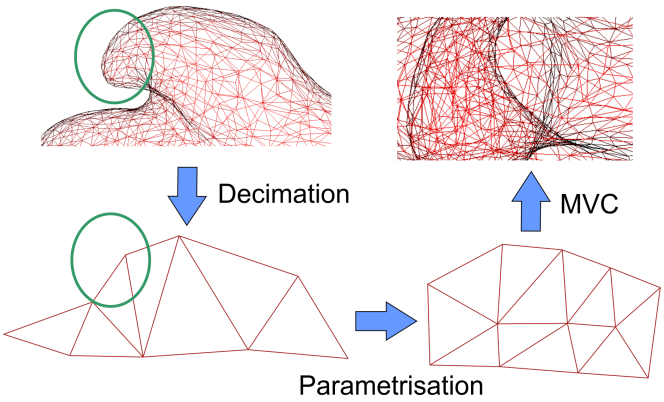


Figure 4.16: Construction of coarse mesh from high-polygonal mesh causes loss of green circled feature. Valid parametrisation (right bottom) of coarse mesh then implies invalid parametrisation of original mesh after reconstruction using mean value coordinates.

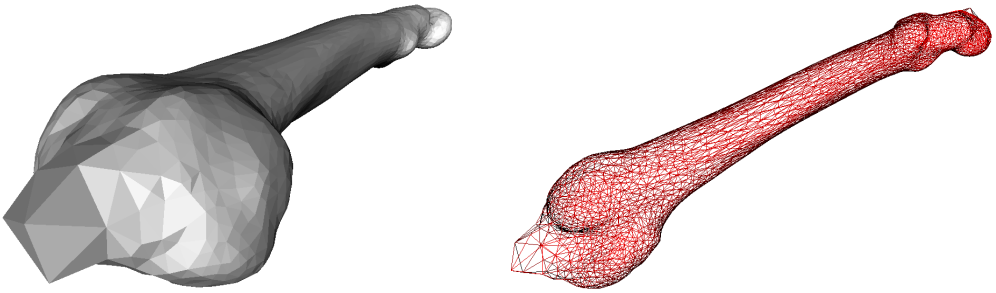


Figure 4.17: Result of two femur bones meshes intermediate weight morph using misaligned parametrisations.

to lie on the same place on parametric domain. Therefore if you look on these two parametrisations merged together in Figure 4.18 you can see, that some distinctive regions with high parametric vertex densities such bone heads lie on different spherical coordinates.

This then causes vertices of supermesh to fall in different regions of both meshes. As an effect of this, supermesh vertices are moved along long paths across model surface during interpolation. You can see visualisation of individual paths in Figure 4.19.

Some of these paths go above the mesh surface in the concave regions and some under the surface in the convex regions of the model. This causes the shape to be distorted. It also means that dense areas of one mesh might fall in the sparse areas of other mesh. Effect of this is seen on the bone heads. There are not

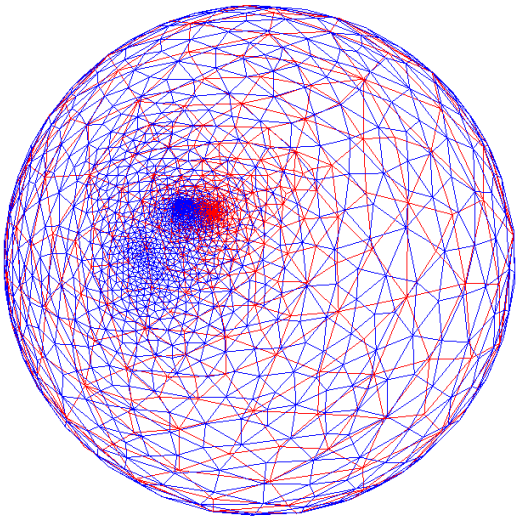


Figure 4.18: Spherical parametrisations of two meshes starting from two aligned femur bones models. Dense areas on pole matches to bone head and should lie on each other.

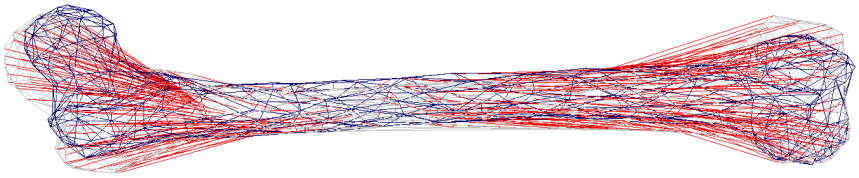


Figure 4.19: Morphing paths (red) between two meshes (blue and grey) based on misaligned parametrisations (Figure 4.18). Number of vertices reduced for better visibility. Incorrect and very long paths results in distorted mesh in Figure 4.17.

enough vertices in the local area of supermesh and the shape of the head cannot be described properly.

To solve this parametrisation alignment problem, parametrisation process must be modified and new step inserted before each re-parametrisation. It will make the parametrisations of the same parts of various input meshes overlap, e.g., dense areas of parametrisation of bone head will be in same polar coordinates in parametric sphere domain for all models of the bone. We will now have to take the parametrisation as complex process that works with all inputs at once. It will now become as follows:

1. Project meshes to sphere to get initial parametrisations

2. Relaxate parametrisations according to Alexa 2000 [1]
3. Find feature points of all mesh that should be aligned
4. Make parametrisations of those points the same by enforcing average value
5. Relaxate parametrisation again but do not move the enforced feature points

I will described added steps in more details now. Although the meshes itself were already aligned, the parametric domains after initial parametrisation are not and therefore the above described problems can occur. Therefore I use approach similar to the [27]. I pick feature points on all input meshes and adjust their parametrisations so that they share shame parametric coordinates.

The main difference is that [27] used user-defined feature points whereas I use automatic selection.

Same as in Section 4.3.1, I use *PCA* to find main axes of individual meshes aligned by non-rigid *ICP*. Then I use same approach to find an inner point. I cast rays to six directions determined by axis starting from the inner centre point (see Figure 4.20). In each direction I find furthest intersected triangle. I then pick the triangle apex closest to the intersection point to be the *feature point*. This way I get six point for each mesh.

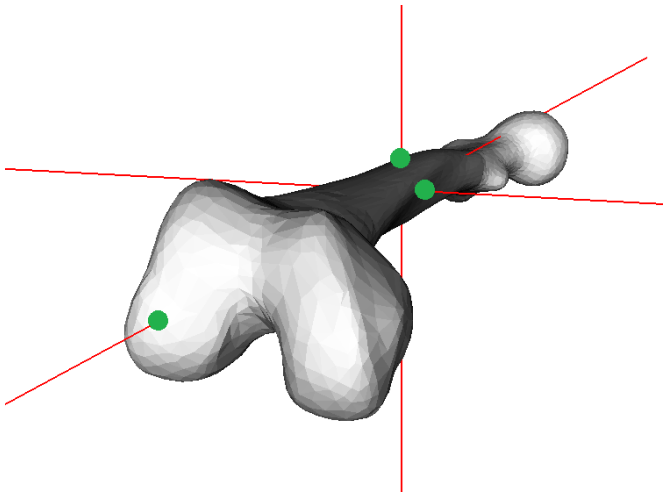


Figure 4.20: Obtaining six feature points of mesh using furthest intersections of main axes going through inner centre point of mesh.

It is important that these six vertices must be unique on each mesh. This limits the bottom of the initial parametrisation coarse mesh size. However we could skip this registration steps for few first iterations if we needed smaller coarse mesh.

Then I align parametrisation points matching to same set of feature points across the inputs. I do this by averaging their parametric coordinates and then normalising the result. It is therefore important that some level of alignment was done with meshes in the beginning of this complete algorithm. This prevents parametric points to lie on opposite sides of parametric domain and therefore to degenerate in the result of averaging.

This step might have caused errors in individual parametrisations. Article [27] then suggests running relaxation process to fix these issues. However I have found this to be insufficient as it is not able to move neighbour vertices for a long path in dense regions.

Therefore I suggest a better approach for distribution of this adjustment across wider area of sphere surface (see algorithm C.14).

This distribution method is very similar to that used in Section 4.4.3. The main difference is in distance measure adapted to spherical space representation and then in normalisation of influence weights as here not all points must be affected by shifts which is difference from the complex mesh deformation application before.

I determine offset of each feature point and then add this offset not only to the respective feature point but also to other point in his wider neighbourhood.

The shift $\overrightarrow{s_i}$ of single feature point $\overrightarrow{x_i}^P$ is calculated using approach described above as

$$\overrightarrow{s_i} = \overrightarrow{x_i}^P - \frac{\sum_{m=0}^{M} \overrightarrow{x}^P_{m,i}}{\left|\sum_{m=0}^{M} \overrightarrow{x}^P_{m,i}\right|} \tag{4.25}$$

where M is number of input meshes and $\overrightarrow{x_m, i}^P$ is parametrisation of feature point matching to $\overrightarrow{x_i}^P$ on mesh m.

I calculate distance $d^P_{i,j}$ of parametrisation point $\overrightarrow{x_j}^P$ to feature point $\overrightarrow{x_i}^P$ as a normalised angular distance of their positions from centre of sphere:

$$d^P_{i,j} = \frac{1}{\pi}\left(\tan^- 1 \frac{|\overrightarrow{x_i}^P \times \overrightarrow{x_j}^P|}{\overrightarrow{x_i}^P \cdot \overrightarrow{x_j}^P|} + \frac{\pi}{2}\right) \tag{4.26}$$

The I use quadratic distribution function $w(i, j)$ to calculate weight of each feature point i to each point j:

$$w(i,j) = \begin{cases} 1 - (\frac{1}{d_{MAX}} \cdot d_{i,j}^P)^u & \text{if } d_{i,j}^P < d_{MAX} \\ 0 & \text{if } d_{i,j}^P \geq d_{MAX} \end{cases}$$

where d_{MAX} is maximum distance for which the weight can be calculated. u is power of distribution influence decrease. I have chosen d_{MAX} to be 0.15 an $u = 1.5$ in my experiments.

Sum of influences on single point cannot be bigger than one. Therefore I use conditional normalising to fix this issue:

$$w_{i,j}^N = \begin{cases} \frac{w_{i,j}}{\sum_i w_{i,j}} & \text{if } \sum_i w_{i,j} > 1 \\ w_{i,j} & \text{if } \sum_i w_{i,j} \leq 1 \end{cases}$$

These formula work even for feature points itself, therefore j can be iterated through entire parametric mesh.

We then calculate adjusted position of $\overrightarrow{\tilde{x}_j}^P$ using formula

$$\overrightarrow{\tilde{x}_j}^P = \overrightarrow{x_j}^P + \sum_{i \in FP} w_{i,j}^N \times \overrightarrow{s_i} \tag{4.27}$$

where FP denotes set of feature points and $\overrightarrow{s_i}$ if shift of i-th feature point.

This way parametrisation is smoothly adjusted to be registered with other input meshes. However there still might be some overlaps, so new relaxation process is then run according to description in Chapter 4.5.3.

This was supposed to result in smooth parametrisation after few iterations only (see Figure 5.14d). There are more measurements and pictures on this topic to be seen in later Section 5.4.2.

In some cases, even this approach might not be robust enough to produce valid parametrisation without overlaps. In this case, I suggest further improvement by iterative distribution of registration adjustment shifts.

In this scenario, total shift of each feature point $\overrightarrow{s_i}$ is divided into several, e.g. 10, parts. Then each minor shift $\overrightarrow{s}_{i,k}$ is distributed among mesh using same approach as described above. Relaxation is then run between each distribution step. Therefore there are much smaller parametrisation errors to be fixed by the relaxation mechanism.

Evaluation of effects of this modification are also discussed in later measurement Chapter 5.4.2.

4.6 Multi-morphing of meshes with reliability maps

At this stage, we have three data sets available. The input meshes with all topological problems solved in Section 4.2 and mutually aligned parametrisations produced in Section 3.2 are the first two. Third input is set of reliability weight vectors for mesh triangles that was secondary product of mesh refinement process in 4.2.

We will now use these data to morph all input meshes together while preferring original input data and suppressing regions created during refinement hole filling step (see algorithm C.5). This way we will combine features of all individual meshes and get one single new mesh. It will also be the final output of the algorithm.

The basic meshing idea was described in theoretical Chapter 3.1 and was mainly inspired by thesis [20]. It consists of three basic steps:

1. Select supermesh among input meshes that will me transformed and returned as the final result. Described in Section 4.6.1.
2. Find barycentric coordinates of each vertex in parametrisation of the supermesh in parametrisations of other meshes. This gives bijective mapping from supermesh to other meshes. More in Section 4.6.2.
3. Use these coordinates to reconstruct the supermesh vertex but replace the parametric domains with the original meshes now. This gives us target points on surface of other meshes that are paired with vertex of supermesh. Described in 4.6.3.
4. Combine those reconstructed point using linear combination to get new position for supermesh vertex. Section 4.6.3.

The output is therefore supermesh with vertex coordinates multi-morphed from surfaces of other meshes. The supermesh is considered to be copy of selected input mesh, therefore it also remains in the "other" mesh set.

4.6.1 Choice of supermesh

First of all, supermesh has to be chosen. Supermesh is term used in thesis [20] and also in theoretical Chapter 3.1 of this document. Regardless of its origin, it describes the mesh graph that is being morphed to various target shapes. If we want good quality of output mesh we have to use good supermesh.

First way how to get a supermesh is usage one of the original input meshes. The second is construction of special mesh using one or more input meshes at once. This was described in Section 3.3.

I decided that the input meshes have enough vertices and edges compared to the complexity of object, because the models have relatively low amount of features and if so, they are shared, so no flat region should be morphed to large peak. Hence there is no need to create supermesh for better description of various target shapes. However if we wanted to improve quality of results, this might be a possible spot for extension.

I assume that all input meshes are reasonably good if we speak about usual metrics like triangle shapes or vertex density distributions. Therefore there are only two differences between them. First one is size of input mesh in number of vertices. We can expect larger mesh to be more likely to fit shape of other mesh after morphing as there is better chance of finding available supermesh vertex for description of local feature. Also some inputs in the very beginning could have more artifacts then the others. Removing artifacts could worsen the topology quality in the major step 2. Therefore mesh with less added triangles is preferred.

Put together, I pick the mesh X_S to be supermesh if

$$\sum_{t \in T_{X_S}} r_{S,t} = \max k \left\{ \sum_{t \in T_{X_k}} r_{k,t} \right\} \tag{4.28}$$

where T_{X_k} is set of triangles of k-th mesh and $r_{k,t}$ is reliability weight of t-th triangle. In Section 4.2.4 it was stated to be 1 for original input triangles and 0 for triangles added during hole filling.

This effectively chooses meshes with most original triangles in absolute count. Such mesh does not have to be perfect and it might even contain more removed artifacts than some smaller mesh. These patched areas will however be replaced in the interpolation phase of morphing.

4.6.2 Barycentric coordinates on spherical domain

Now parametrisation of the supermesh must be expressed in surface space of parametrisations of other meshes. Barycentric coordinates $\overrightarrow{\Lambda_i} = (\lambda_0, \lambda_1, \lambda_2)$ are used to express coordinate of each supermesh parametrisation vertex $\overrightarrow{s_i}^P$ using barycentric coordinates in the triangle $\overrightarrow{T} = (a, b, c)$ of mesh X_k in which $\overrightarrow{s_i}^P$ lies when both parametric spheres are aligned to same centre:

$$\overrightarrow{s_i}^P = \lambda_0 \cdot \overrightarrow{x}^P_{k,a} + \lambda_1 \cdot \overrightarrow{x}^P_{k,b} + \lambda_2 \cdot \overrightarrow{x}^P_{k,c} \tag{4.29}$$

If we treated parametrisations like triangular meshes in traditional way, then barycentric coordinates of point $\overrightarrow{p}$ inside triangle $\overrightarrow{x_a}$, $\overrightarrow{x_b}$, $\overrightarrow{x_c}$ would be found by solving equation

$$\begin{aligned}
\overrightarrow{p} &= \lambda_0 \cdot \overrightarrow{x_a} + \lambda_1 \cdot \overrightarrow{x_b} + \lambda_2 \cdot \overrightarrow{x_c} \\
\overrightarrow{p} &= \begin{bmatrix} \overrightarrow{x_a} & \overrightarrow{x_b} & \overrightarrow{x_c} \end{bmatrix} \cdot \begin{bmatrix} \lambda_0 \\ \lambda_1 \\ \lambda_2 \end{bmatrix} \\
\overrightarrow{p} &= \begin{bmatrix} x_{a,0} & x_{b,0} & x_{c,0} \\ x_{a,1} & x_{b,1} & x_{c,1} \\ x_{a,2} & x_{b,2} & x_{c,2} \end{bmatrix} \cdot \begin{bmatrix} \lambda_0 \\ \lambda_1 \\ \lambda_2 \end{bmatrix}
\end{aligned}$$

This is non-homogenous matrix equation that has single solution if the triangle does not have zero area and if the point $\overrightarrow{p}$ lies in the plane of triangle. This is however not true on the spherical domain. The triangles are not flat here. Their edges are not lines but parts of great circle. Therefore points on their surfaces do not lie on plane determined by their three points (see Figure 4.21).

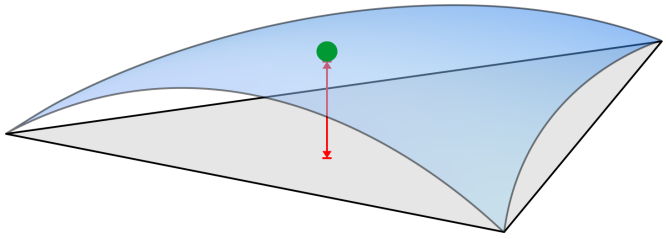


Figure 4.21: Difference (red) between spherical triangle (blue) and planar triangle (grey) when barycentric coordinates of point on sphere (green) are calculated.

Therefore usage of planar barycentric relations will cause points to be evaluated lying outside of the triangle. We might add some tolerance to the test, but the error will get larger with size of triangle. Thesis of Ing. Parus Ph.D. [20] suggests normalisation of barycentric coordinates gained for planar triangle to get barycentric coordinates in spherical triangles and points on unit sphere domain:

$$\overrightarrow{\Lambda}^S = \overrightarrow{\Lambda} \cdot \frac{1}{\lambda_0 + \lambda_1 + \lambda_2} \tag{4.30}$$

I tested this approach and results in Section 5.5.1 shows that this fixes the algorithm so that correct barycentric coordinates inside single unique spherical triangle is found for each vertex of supermesh parametrisation.

From time complexity point of view, each point of supermesh is tested against each triangle of each other mesh to find the spherical triangle where it lies and appropriate barycentric coordinates. This then leads to quadratic complexity $O(M \cdot N^2)$ if M is number of inputs, all inputs have similar vertex size N and number of triangles is linear in vertex count.

This might be a lot for large meshes and therefore some smarter approach to find triangle for point could be used. The coordinates on spherical domain are 2D thanks to the locked unit radius, so quad-tree or 2D grid can be used to speed up triangle location instead of 3D structures possibly gaining larger speed up.

If we use one of the inputs as the supermesh, then we can easily find triangles and barycentric coordinates in the same mesh as $(1, 0, 0)$. This allows unified implementation of later steps. As there are always at least three triangles sharing each point in closed mesh, I pick the one with highest reliability. This prevents the vertex to be disabled in interpolation process if some of its triangles is unreliable.

4.6.3 Interpolation in AMS

Now we have spherical barycentric coordinates $\overrightarrow{\Lambda}^S_{i,k}$ for each parametric vertex of supermesh $\overrightarrow{s_i}^P$ and each mesh k along with identifications of respective other mesh triangles.

For each mesh k, the parametric vertices of triangle $\overrightarrow{T} = (a, b, c)$ can now be replaced by original mesh vertices $\overrightarrow{x}_{k,a}$, $\overrightarrow{x}_{k,b}$, $\overrightarrow{x}_{k,c}$. This will give us approximation of local surface by supermesh vertex $\overrightarrow{s_i}$:

$$\overrightarrow{s}_{i,k} = \overrightarrow{\Lambda}^S_{i,k} \cdot (\overrightarrow{x}_{k,a}, \overrightarrow{x}_{k,b}, \overrightarrow{x}_{k,c}) \tag{4.31}$$

The approximation is perfect at point $\overrightarrow{s_i}$ but the supermesh gets further from surface X_k inside the supermesh triangles if some vertices of X_k are not covered by supermesh vertex (see Figure 4.22).

This is the reason for picking the largest mesh for supermesh and why construction of even richer mesh could improve results.

This way we get approximations of surfaces of all input meshes using supermesh vertices $\overrightarrow{s_i}$ as sets of vertices $\overrightarrow{s}_{i,k}$ for mesh k.

Each such projection of point $\overrightarrow{s_i}$ to mesh k then represents reference point in *Affine Morphing Space* (*AMS*) as stated in [20] and explained in Section 3.6.2.

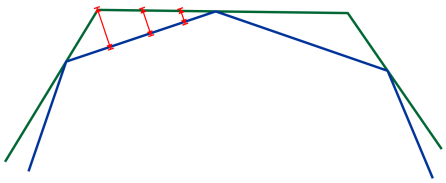


Figure 4.22: Limit approximation of target mesh (green) by supermesh (blue). Approximation error (red) is zero at supermesh vertices but increases in the middle of its triangles.

It means that we now have set of points $\overrightarrow{s_i}$ and we can use morphing weights as barycentric coordinates inside. This way we calculate linear interpolation between them provided that sum of barycentric coordinates is 1.

If no additional information was provided, we would pick the middle point in this space as final output using uniform linear coefficients $w_0 = w_1 = ... = w_M = \frac{1}{M}$ where M is number of inputs.

However this is where the reliability weights of input mesh triangles come to play. I build the interpolation coordinates from the reliability of triangles T_k, j where the projection of supermesh parametric vertex $\overrightarrow{s}^P_{i,k}$ was found to lie in individual meshes X_j. Therefore if triangle T_k, j was untouched by artifact removal process, the initial linear coordinate $l_{i,k}$ will be 1. If this was added while filling holes it will have zero and will not influence the result.

The final weights are then normalised to add up to 1 as the extrapolation is not desired.

Only one special case must be handled. It might happen that all projections lie in added triangles with zero reliability. Then uniform coordinates are used again giving unreliable results as there is no more information to use.

Put together, linear coordinate $l_{i,k}$ of final output vertex $\overrightarrow{r_i}$ in AMS for k-th projection of supermesh vertex $\overrightarrow{s_i}$ are given using:

$$l_{i,k} = \begin{cases} \frac{r_{k,T_k}}{\sum_h^M r_{h,T_h}} & \text{if } \sum_h^M r_{h,T_h} > 0 \\ \frac{1}{M} & \text{if } \sum_h^M r_{h,T_h} = 0 \end{cases}$$

where T_k is triangle of mesh k where the supermesh vertex projection lies in and r_{k,T_k} is its reliability. M is number of inputs.

i-th vertex $\overrightarrow{r_i}$ of output mesh can be found using these linear coefficients $l_{i,k}$ and AMS determined by vertices $\overrightarrow{s}_{i,k}$:

$$\overrightarrow{r_i} = \sum_{k}^{M} l_{i,k} \cdot \overrightarrow{s}_{i,k} \tag{4.32}$$

Final result is mesh R with vertex coordinates $\overrightarrow{r_i}$ and topology described by supermesh S. It is a mesh that combines features of all input meshes. Its shape should be mix of their shapes. However the Figure 4.23 shows that it is not always true.

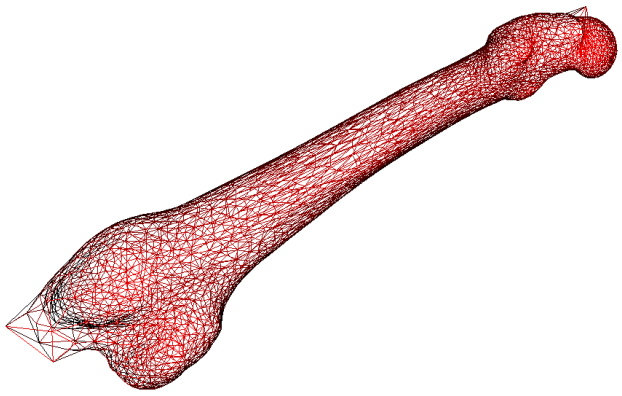


Figure 4.23: Output of multi-morphing step as step 5 of spherical domain version of the method applied to femur bone.

4.7 Direct on-mesh alternative for morphing

Now we will get to the other variant of the main algorithm. As far we spoke about spherical domain morphing, but now we will discuss direct on-mesh alternative as was mentioned in the very beginning of Chapter 4. Therefore, this section does not continue from the previous one, but completely replaces steps in Sections 4.5 and 4.6 instead. It directly continues where the Section 4.4 ended. Therefore no parametrisation is needed and the morphing is done on non-rigidly aligned meshes instead.

Morphing using parametrisation on spherical domain works well if the parametrisation step was able to produce valid parametrisation without triangle overlaps and at the same time preserve mutual registrations of features in all meshes.

However as experiments in Chapter 5.4.3 show, it is often not easy to provide both valid and mutually aligned parametrisation. This then causes the product of morphing to be visually invalid and distorted (see Figure 4.17).

Although I have put large portion of my effort into attempts to solve this problem conventionally on the spherical domain, it seems that it might not be possible or efficient. I was not even able to find any solution as all earlier discussed methods simply assumed that no such problems will occur.

One possible solution for misalignment of parametrisations would be their merge to get rich supermesh candidate that would be able to preserve all details in interpolation extremes. However the long distances between extreme poses would anyway inevitably cause shape corruption in all inner morphing phases.

Therefore I have come with completely different morphing schema that can be used if spherical parametrisation fails.

I exploited the fact that non-rigid *ICP* method gave us well aligned models and I simply skipped projection to sphere phase and used those models directly as parametric domains.

This means that no parametrisation step has to be done at all.

Supermesh is once again chosen to be the most reliable of parametric domains, this time of deformed meshes $\tilde{X}_k$.

Of course there still are some minor differences between the outliers of individual deformed models. However they should be now small enough so that for each vertex $\overrightarrow{s_i}$ of supermesh S it is possible to find its counter pair point on the surface of other mesh $\tilde{X}_k$ by minimisation of their mutual distances.

This means that for each vertex $\overrightarrow{s_i}$ and each of deformed input meshes $\tilde{X}_k$, nearest point $\overrightarrow{s}_{i,k}$ is found as nearest point on nearest triangle $T_{i,k}$:

$$T_{i,k} = \arg\min_{T_j \in \tilde{X}_k} distance(\overrightarrow{s}_i, T_j) \tag{4.33}$$

Distance of point to triangle can be determined by distance to its plane ρ if point's perpendicular projection $\overrightarrow{x}^\rho$ falls into the area of triangle or distance to nearest edge of triangle otherwise:

$$distance(\overrightarrow{x}, T) = \begin{cases} |\overrightarrow{x} - \overrightarrow{x}^\rho| & \text{if } \overrightarrow{x}^\rho \text{inside} T \\ \min_{e_i \in T} distance(\overrightarrow{x}, e_i) & \text{otherwise} \end{cases}$$

General barycentric coordinates in 3D discussed in 4.6.2 can be used to test mutual position of point and triangle directly in 3D space. Rotation of all points

to plane of any two standard axes and usage of 2D barycentric coordinates is second alternative.

Distance of point to edge of triangle is told by distance of perpendicular projection of point $\overrightarrow{x}^e$ to its line if the projection lies between end points $\overrightarrow{A}$ and $\overrightarrow{B}$ or distance to the nearer one of them otherwise:

$$distance(\overrightarrow{x}, e) = \begin{cases} |\overrightarrow{x} - \overrightarrow{x}^e| & \text{if } \overrightarrow{x}^w \text{between} \overrightarrow{A} \text{and} \overrightarrow{B} \\ \min_{\overrightarrow{p} \in \{\overrightarrow{A}, \overrightarrow{B}\}} |\overrightarrow{x} - \overrightarrow{p}| & \text{otherwise} \end{cases}$$

Then I use nearest triangle $T_{i,k}$ and its nearest point to find its barycentric coordinates $\overrightarrow{\Lambda}_{i,k}$. Barycentric coordinates also helps to distinguish 3 possible outcomes shown above:

1. $\max \overrightarrow{\Lambda}_{i,k} = 1$ - nearest point is vertex of $\tilde{X}_k$
2. $\min \overrightarrow{\Lambda}_{i,k} = 0$ - nearest point lies on edge of $\tilde{X}_k$
3. otherwise - nearest point lies inside of triangle of $\tilde{X}_k$

This helps to improve the guess of projection triangle reliability. If the nearest point is vertex of $\tilde{X}_k$, we can pick any of its triangles to be used as source triangle for barycentric coordinates. We will therefore pick the one with highest reliability weight. Similarly with nearest point on edge, we can pick the most reliable triangle of edge.

This makes sure that the mesh $\tilde{X}_k$ will have influence on target position of $\overrightarrow{s_i}$ through its nearest point $\overrightarrow{s}_{i,k}$.

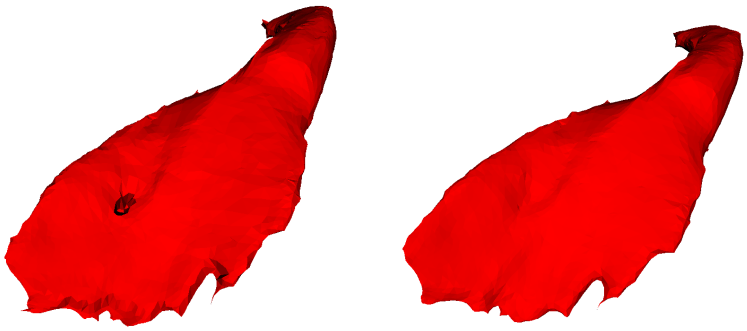
The same improvement could also be applied on previously discussed spherical parametrisation morphing.

I also added similar restriction to nearest triangle search as in non-rigid *ICP* step. I enforce the nearest triangle j to have normal angle difference compared to morphed point's normal i not higher than 90 degrees:

$$\overrightarrow{n_i} \cdot \overrightarrow{n_j} \geq 0$$

This once again prevents false target surface pick in thin mesh surface areas (see Figure 4.24).

As we now have triangle and barycentric coordinates $\overrightarrow{\Lambda}_{i,k}$ for each supermesh vertex $\overrightarrow{s_i}$ and each mesh $\tilde{X}_k$, we can go back to Chapter 4.6.3 and proceed with interpolation as all following steps are identical.



(a) Normal check OFF (b) Normal check ON

Figure 4.24: Influence of normal check in nearest triangle search in direct morphing. See the hole in the flat area of Iliacus muscle on left image. The opposite surface would be evaluated closer than the correct one if no restriction was used.

The only difference is therefore in change of spherical parametric domain to non-rigidly aligned deformed input meshes and thus change of spherical barycentric coordinates to general 3D barycentric coordinates for nearest point in nearest triangle on the target domain (see algorithms C.5 and C.6 for comparison).

The coordinates used for morphing are still being read from original non-deformed input meshes so no information was lost prior to morphing. The deformed and non-rigidly aligned meshes were only used as source of mutual surface positions. In another words they create navigation maps to find sources of coordinates for the morphing.

Finally, the supermesh is again morphed so that it combines features of all input meshes. As the Figure 4.25 shows, the similarity is now significantly improved. The results are discussed and compared with spherical domain approach in Chapter 5.5.2.

Figure 4.25: Output of multi-morphing step as step 4 of direct on-mesh version of the method applied to femur bone.

4.8 Summary

4.8.1 Overview

I have provided detailed description of two versions of an algorithm that is able to take several various meshes with possible non-manifold artifacts or small holes on the input and produce one closed mesh without non-manifolds on the output that is an approximation of shapes of all.

The algorithm flow is described in Figure 4.1 for the version with spherical parametric domains and on 4.2 for version with direct on-mesh morphing. Main steps of algorithms are summarised in algorithms 4.1 and 4.2. Individual steps are then described more in referenced sub-algorithms published in appendix C.

Algorithm 4.1 Complete overview of algorithm with spherical parametric domain

1: Remove non-manifolds and fill holes {more in alg. C.1}
2: Align meshes using *PCA* {more in alg. C.2}
3: Align meshes non-rigidly using modified *ICP* {more in alg. C.3}
4: Find spherical parametrisations {more in alg. C.4}
5: Multi-morph meshes using parametrisations {more in alg. C.5}

Algorithm 4.2 Complete overview of algorithm with direct on-mesh parametrisation

1: Remove non-manifolds and fill holes {more in alg. C.1}
2: Align meshes using *PCA* {more in alg. C.2}
3: Align meshes non-rigidly using deformation by modified *ICP* {more in alg. C.3}
4: Multi-morph meshes directly using non-rigidly aligned models {more in alg. C.6}

4.8.2 Asymptotic analysis

Analysis shows that all elementary steps are linear in number of mesh inputs M and linear or quadratic in size of individual meshes N. Registration on vertex to vertex or vertex to triangle base is usually reason for quadratic behaviour. In second case, I assume the number of triangles to be linear to number of vertices for purpose of this analysis which is valid for genus 0 closed surface meshes. I also assume all meshes to have the same size N.

Mesh refinement step

Artifact removal is run for M meshes. Then removing edges is linear in number of edges. Although in general there are N^2 edges in a graph of N points, we can state that there are only $O(N)$ edges in each mesh after we agreed on $O(N)$ triangle limit. Therefore non-manifold edges are removed in $O(N)$ steps.

Similar applies to vertex neighbour tests in vertex filtering sub-step. Again, general node in graph has $O(N)$ neighbours but we can expect that it there will be some constant boundary in usual surface meshes, although exceptions might be found like central vertex of cone model. Nevertheless, let each vertex be processed in $O(1)$ leaving complete mesh be processed in $O(N)$.

Finding mesh components for solitary mesh part detection then requires $O(E)$ steps where E is number of edges. We already stated $E = O(N)$ hence this step is also linear $O(N)$.

Last part is mesh hole filling. Ready to use implementation of *VTK* was used and it does not state the complexity. However we can assume that at least $O(E) \sim O(N)$ steps are required that finds the holes. Than size and number of holes influence final complexity. Let's assume that are only few small holes relative to N so that hole filling step can be left with $O(N)$ total complexity.

This way the complete mesh refinement step is done in $O(M \cdot N)$ time.

Initial registration step

The mesh alignment step is set of $M - 1$ alignment processes. Each consists of $O(N)$ main axes calculation and 4 $O(N^2)$ error distance calculations. This can however be reduced by smarter data structure like kd-tree or uniform grid. If binary search is applied, the final complexity is $O(N \log N)$. Therefore final complexity of alignment step is $O(M \cdot N^2)$ with simple implementation or $O(M \cdot N \log N)$ with optimised structures.

Non-rigid alignment step

Then non-rigid *ICP* deformation step is problematic as it contains iterative process with undetermined number of iterations.

For each of M meshes feature points are chosen based on N. However there is a constant limit for the size of selection and therefore only c_0 points are chosen. Each pick then follows selection of points in neighbourhood.

It involves constant number of iterations each picking neighbourhoods of points from previous iteration. We already stated that each point can only have constant

maximum number of neighbours. This means that each neighbourhood is created in $O(c_1 \cdot c_2) \sim O(1)$ steps.

Then *ICP* for each region consists of unknown number of iterations. However thanks to the iteration limit there is some c_3 that bounds the complexity so that *ICP* is done in $c_3 \cdot O_{iter}$.

Each iteration contains finding nearest points for c_1 points in region. This can again be done in $O(c_1 \cdot N)$ using brute force or $O(c_1 \log N)$ using binary search structures.

Then c_0 deformations are distributed to mesh using $O(N)$ algorithm.

To sum it up, the final complexity for single mesh is $O(c_0 \cdot (c_1 \cdot c_2 + \cdot c_3 \cdot O_{iter}))$ which then determines complexity of steps to be $O(M \cdot N)$ or $O(M \cdot \log N)$ depending on the implementation. However in practise, left out constants are very large and therefore this step takes same or more time than the others for usual meshes with few thousands vertices (see Chapter 5.6.2 for details).

Spherical parametrisation step

Parametrisation consists of coarse mesh hierarchy construction and individual relaxation iterative processes.

For of M meshes $O(\log_2 N)$ coarse meshes are constructed. Each of them has $O(N)$ vertices. Calculation of *Mean value coordinates* then requires $O(N_i \cdot N_{i+1}) \sim O(N^2)$ operations. Same complexity applies to reconstruction of higher mesh later on.

Hence cascade hierarchy construction has $O(N^2 \cdot \log N)$ complexity.

It might however be possible to further optimise *MVC* calculations using above described space search structures if only close triangles would be allowed to be evaluated although it's not used in original source article [24]. This might reduce total complexity to $O(N \cdot \log^2 N)$ but might require some more adjustments. It would also mean that mesh reconstruction using *MVC* would have to be improved to fit new complexity boundary. This might not be possible so I stick with the higher guess.

Then for each coarse mesh iterative relaxation is run. It consists of maximum c iterations where c is up to 10 000. Each iteration is linear in number of vertices and their neighbours. We have already stated that each vertex has only constant number of neighbours and therefore each iteration is $O(N)$. This then leaves relaxation process to have complexity of $O(N)$. The relaxation is run on each cascade level, hence it adds $O(N \cdot \log N)$ calculations.

The final parametrisation cost is then $M \cdot O(N^2 \cdot \log N) + O(N \cdot \log N) \sim O(M \cdot$

$N^2 \cdot \log N$).

If the cascade schema is omitted, the complexity is reduced only to cost of projection and relaxation which makes it $O(N)$ total when ignoring relatively high iteration count constant c.

If the direct morphing alternative was used then parametrisation step would be skipped.

Multi-morphing step

In multi-morphing phase, supermesh is picked based on sum of reliability weights in $O(M \cdot N)$ time. Then each of N supermesh's vertices is calculated using barycentric coordinates in $M - 1$ other meshes.

For each such vertex and mesh, barycentric coordinates of parametric image in parametrisation of other mesh has to be found. That includes testing of $O(N)$ triangles according to former agreement that number of triangles is linear to number of vertices. Again, special structures can be used to reduce this search to $O(\log N)$. Other calculations are then performed in constant time.

If the direct morphing approach was used, then all steps would be analogous and identical in terms of asymptotic analysis. Same applies to possible data structure based enhancement.

Therefore total cost if multi-morphing is expressed as $O(M \cdot N^2)$ or $O(M \cdot N \cdot \log N)$ depending on data structure used.

Summary

If we sum all individual complexities for spherical parametrisation version we get

$$\begin{aligned} O(M \cdot N) + O(M \cdot N^2) + O(M \cdot N) + \\ O(M \cdot N^2 \cdot \log N) + O(M \cdot N^2) \sim \boldsymbol{O(M \cdot N^2 \cdot \log N)} \end{aligned} \tag{4.34}$$

or

$$\begin{aligned} O(M \cdot N) + O(M \cdot N \log N) + O(M \cdot \log N) + \\ O(M \cdot N^2 \cdot \log N) + O(M \cdot N \cdot \log N) \sim \boldsymbol{O(M \cdot N^2 \cdot \log N)} \end{aligned} \tag{4.35}$$

depending on data structure used to search nearest points and triangles of mesh in 3D-space.

However as can be seen, the final complexity $O(M \cdot N^2 \cdot \log N)$ in case of cascade schema spherical parametrisation is same for both brute force and binary search implementation as it is determined by the MVC calculation and later reconstruction part of parametrisation step. If we were able to optimise this, we could hope to get $O(M \cdot N \cdot \log^2 N)$ instead.

If direct morphing on mesh domain technique was used then no parametrisation would be needed saving us the most costly part of pipeline and giving the final time complexity as $O(M \cdot N^2)$ or $O(M \cdot N \log N)$ in the case of effective data structures for search of points in triangle meshes.

Once again I have to emphasise that this estimations are only valid if the assumptions made are maintained. Those were that both number of triangles and number of edges of the mesh is linear to number of vertices and degree of each vertex is limited by some constant independent on mesh size. These properties were observed on real data and are likely to be maintained by any reasonable closed genus 0 triangular surface mesh, but they are not at all valid for general graph where number of edges can be $O(N^2)$ and degree of all vertices $O(N)$.

Chapter 5

Results

5.1 Implementation

I implemented the method described in Chapter 4 using the integrated development environment *Microsoft Visual Studio 2010 Ultimate x64* and programming language *C++*.

I have divided the solution to two parts. The first one in project *MeshRegister* contains only the algorithm itself. It heavily uses *VTK* framework (see Section B.1.2) for data structure, their manipulation and processing. The main class implementing the algorithm *vtkMeshRegister* is inherited from abstract class *vtkPolyDataAlgorithm* so it can be used by other SW as a standard *VTK* filter. The main "component" relations can be seen on schematic Figure 5.1.

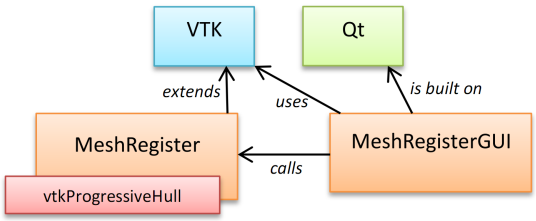


Figure 5.1: Schema of application parts and dependencies. The orange blocks are my projects. Red block is integrated code. Other blocks are independent public frameworks.

The rest of implementation is in project *MeshRegisterGUI*. It contains the *GUI* that allows to control the application, specify inputs and observe and store outputs. There were two major reasons for such division. First, it makes it easier to move only the filter to another application. Second, it uses *Qt* framework for

user interface (see B.1.1). It is a huge package of several hundred megabytes that has to be downloaded and compiled [1]. Therefore I aimed for isolation of filter from its code to avoid possible dependencies.

More detail information about implementation can be found in Chapter B.2. Description of application architecture and individual classes is then available in Chapter B.3.

I also worked with *Borland Delphi 7* IDE. I used it to modify the *GUI* only parametrisation application *Embedding3D* of Ing. Parus Ph.D. (see B.1.4) to a DLL library which can be called by my *C++* code. For this reason I had to renew my high-school knowledge of *Pascal* language as the original code was provided in *Delphi*. I also created appropriate *C++* code in class *Embedding3D* for comfortable usage of *DLL*. I had to modify both interface method of the library as well as the logic to provide access to parametrisation method and enable possibility to specify my own centre of projection.

Almost all code is platform independent. That includes *VTK* and *Qt* libraries. Only time measurement in *MyTimer* and DLL loading in *Embedding3D* uses *Windows API*. The first one could easily be modified and the second fully removed. However cross-platform portability was not a goal.

The testing during development and in the final phase was done using data provided by my supervisor. They consisted from both latest already filtered data and former corrupted data I used for my bachelor thesis.

The meshes itself were processed data from medical scanners and three different sources. I did most of the development phase using *Femur bone* as it provides distinctive shape and smooth surface, so it is easy to observe effects of modifications.

I did not have an alternative for my work as a reference solution, so I run only partial comparisons of parametrisation and non-rigid alignment phase. I used combination of ready to use implementations available and my own reimplementations based on article descriptions.

5.2 Experimental setup

I have run various experiments using input sets with both manifold and non-manifold meshes. Data were provided by supervisor.

Manifold only data sets:

[1] There is a binary version of *Qt* available. However it supported only *Visual Studio* of version 2008 when I started implementation. Therefore compilation from source code is required for proper integration with *Visual Studio 2010*.

- *Femur bone* ... three models, two with 2 502 vertices one with 42 502 vertices
- *Iliacus muscle* ... two models, one with 3 248, second with 3 978 vertices
- *Sartorius muscle* ... three models, 2 390, 9 004 and 4 956 vertices

Non-manifold and mixed data sets:

- *Femur bone* ... two smaller models from manifold set modified using my *vtkDamage* filter to randomly add 51 and 25 non-manifold edges, 146 and 73 non-manifold vertices, 92 and 53 holes[2] an 14 isolated components each. The results are total vertex counts 2 698 and 2 629.
- *Sartorius muscle* ... one manifold model from manifold set with 9 004 vertices and one non-manifold model 1.1 from my bachelor work [13] with 9 001 vertices, 2 non-manifold edges and 11 isolated components.

I have used both artificially and naturally damaged meshes as I did not find suitable natural candidate that would have all artifacts I needed for all tests.

The tests were run on consumer PC with quad core CPU *Intel Core i7 2600K* at 3.5GHz with *Hyper-Threading*, 16 GB of dual channel 1600 MHz DDR-III RAM and *Windows 7 x64* OS. GPU acceleration using *CrossFire* configuration of two *AMD Radeon HD 6870 1GB* graphic cards were used in standard *VTK*'s *OpenGL* based renderer and in one reference implementation of spherical parametrisation used in tests [2].

5.3 Alignment

5.3.1 Alignment using original and course mesh

I measured accuracy of *PCA* based alignment with full size and coarse mesh with 300 vertices. The coarse mesh here is the outer hull gained using filter described in Section B.1.3. Example of such outer hull can be seen in Figure B.5.

The Table 5.2 presents measured times and distances between meshes. The distance error was measured on point-to-point basis and denotes average value for multiple inputs.

[2]Counting holes in non-manifold mesh is not precise as there is no definition what is inside and what outside. The hole removal process relies on fixing of non-manifolds first.

Input	# of meshes	# of vertices	Coarse mesh			Full-size mesh	
			Time [ms]		Error (avg)	Time [ms]	Error (avg)
			Align only	"+ coarse"			
femur	2	2502 and 2502	888	2 165	36.5771	55 002	36.5771
illiacus	2	3248 and 3978	881	4 895	58.5297	112 458	58.5297
sartorius	3	2390 and 9004 and 4956	1 801	7 592	44.9223	301 252	44.9223

Figure 5.2: Comparison of initial alignment precision and execution time based on working mesh used. "+coarse" time includes coarse mesh construction. Errors are always measured on the final full size mesh for the purpose of the test.

As the results show, there is no measurable difference between alignment on full-size mesh and coarse mesh. The coarse mesh is used only to accelerate the *PCA* and it seems to have enough information to describe direction trend so the error calculated on the full size mesh after the alignment is finished does not differ from the full mesh approach. However the execution times differ a lot as no acceleration data structure was used. Figure 5.3 confirms this result visually as there is no difference observable.

(a) On coarse mesh (b) On full size mesh

Figure 5.3: Three Sartorius muscle models aligned using PCA on coarse and full-size mesh.

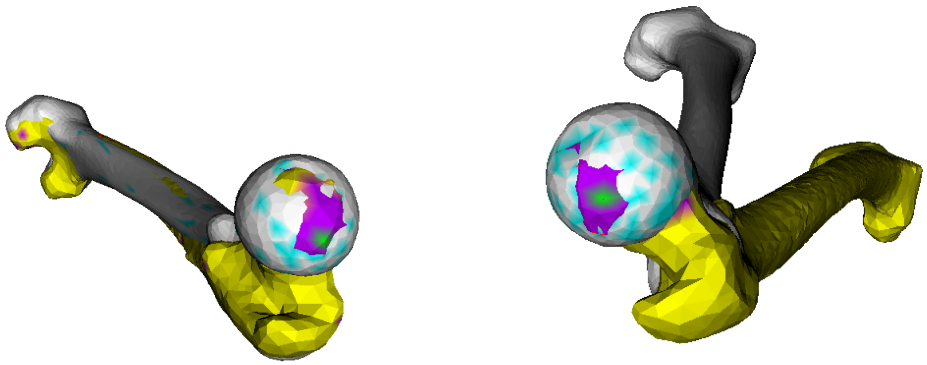
The conclusion here is that usage of coarse mesh is reasonable and does not bring any measurable loss of accuracy while saving significant execution time ($O(N^2)$ without optimisation).

5.3.2 Region selection in ICP

I have compared the original proposal of region points selection from [5] and my own modification based on 3-neighbourhood. The goal was to find a method that better describes local shape of mesh.

Figures 4.10 and 4.11 compare selection of points used for region.

Figure 5.4 then compares final alignments gained from *ICP* run on both selections of region for the same centre vertex in head of femur bone.



(a) Based on [5] (b) 3-neighbourhood

Figure 5.4: Final transformation of source mesh (yellow) to target mesh (white) according to local ICP result for region (purple) of single feature point (green) on bone's head. Various region sampling mechanisms.

It is clear that the 3-neighbourhood better describes the local shape and gains closer local alignment.

5.3.3 Alignment comparison

I have compared final alignment quality and also execution times of *PCA*, rigid *ICP* and non-rigid *ICP* with both region selection methods. Method "A" denotes region selection based on [5], method "B" my proposal of 3-neighbourhood. I have measured the distance of meshes after alignment to evaluate alignment quality. I have used two smaller models of Femur bone from manifold input set for this experiment. Hence both inputs have 2 502 vertices.

Table 5.5 summarises average values gained. Times for *ICP* based methods do not include preprocessing by *PCA*. Non-rigid *ICP* "A" uses 150 regions with 100 vertices each. Version "B" uses 500 regions with variable number of vertices

depending on 3-neighbourhood size. I assume that total number of vertices used in comparison is in both cases approximately 1 500. However it seems, that adding more regions to the method "A" does not significantly reduce the final error.

Method	Time [ms]	Error [-]
PCA	846	36,5771
Rigid ICP	1 081	20,4195
Non-rigid ICP A	51 436	18,1376
Non-rigid ICP B	3 845	11,6893

Figure 5.5: Comparison of various alignment methods. Error measured as average distance between nearest points taken from both perspectives.

Results proves 3-neighbourhood (noted as non-rigid method "B") to be globally more successful strategy. Hence I use this approach in rest of measurements.

Figures 5.6 and 5.7 further document this fact by direct comparison of alignment in complicated regions of mesh.

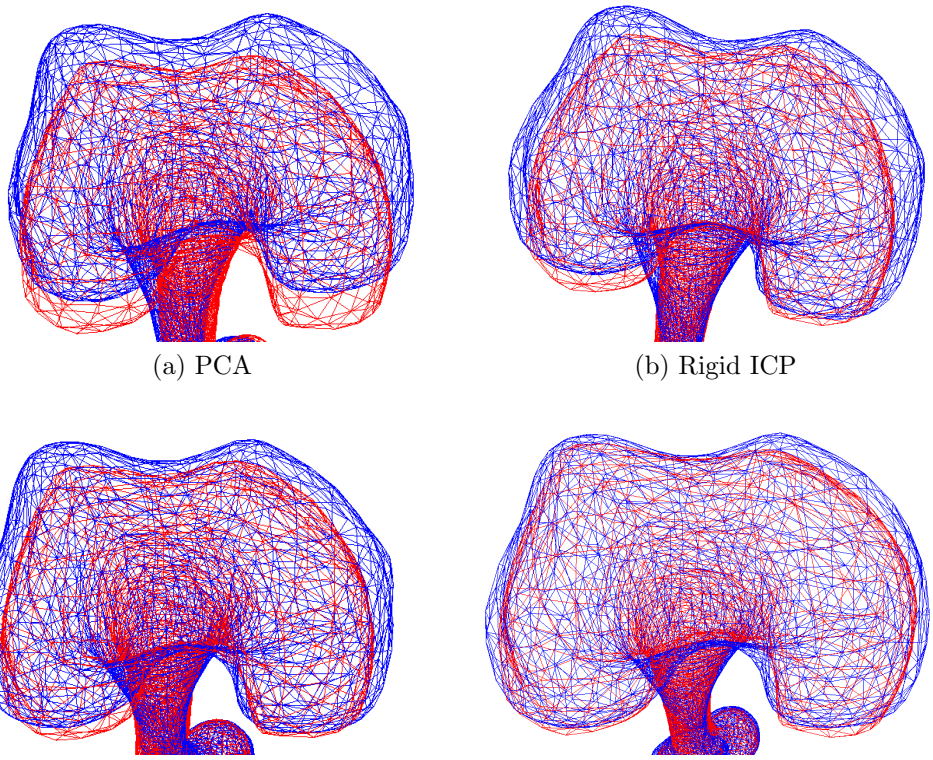
Although increasing number of feature point regions for non-rigid *ICP* improves local alignment, using all points of original mesh as is demonstrated in Figure 5.8 causes distortion as no interpolation is then possible in deformation step and therefore sharp changes occur. Therefore we always use only 10% but at most 500 feature vertices to create *ICP* regions.

5.4 Parametrisation

5.4.1 Initial spherical parametrisation

This section contains results of experiments with spherical parametrisation run on a single mesh only. The goal of this was to find the best spherical parametrisation strategy which will be later extended to multi-mesh setup.

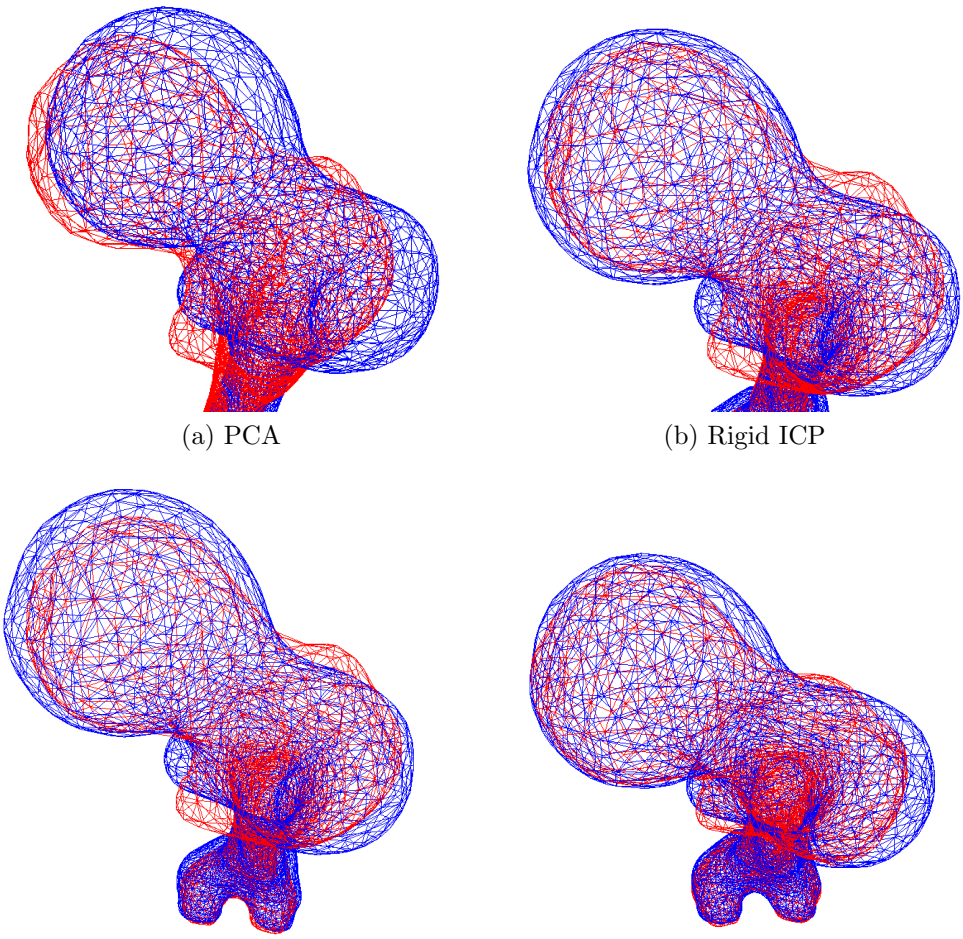
I have used the high-resolution 42 501 vertices model of femur bone from manifold input set. It provides good challenge for all tested parametrisation techniques because they tend to fail in producing valid parametrisation although the mesh contains visually good topology and no artifacts. Figure 5.9 demonstrates both original bone and initial parametrisation given by projection to sphere.



(a) PCA (b) Rigid ICP

(c) Non-rigid ICP "A" (d) Non-rigid ICP "B"

Figure 5.6: Comparison of various methods for alignment source mesh (red) to target mesh (blue). Bottom head of Femur bone.



(a) PCA (b) Rigid ICP

(c) Non-rigid ICP "A" (d) Non-rigid ICP "B"

Figure 5.7: Comparison of various methods for alignment source mesh (red) to target mesh (blue). Top head of Femur bone.

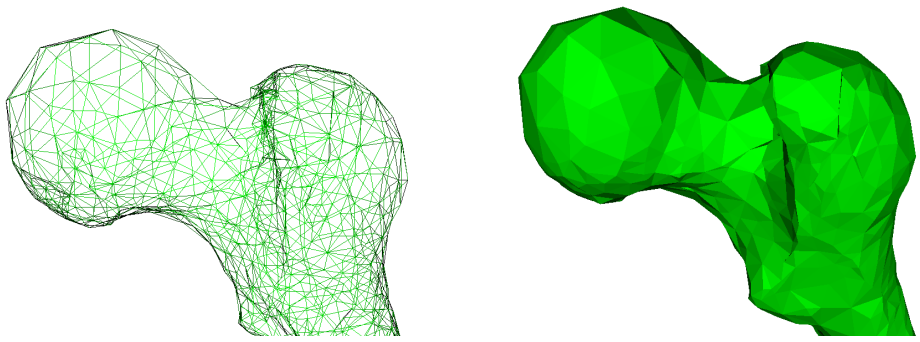
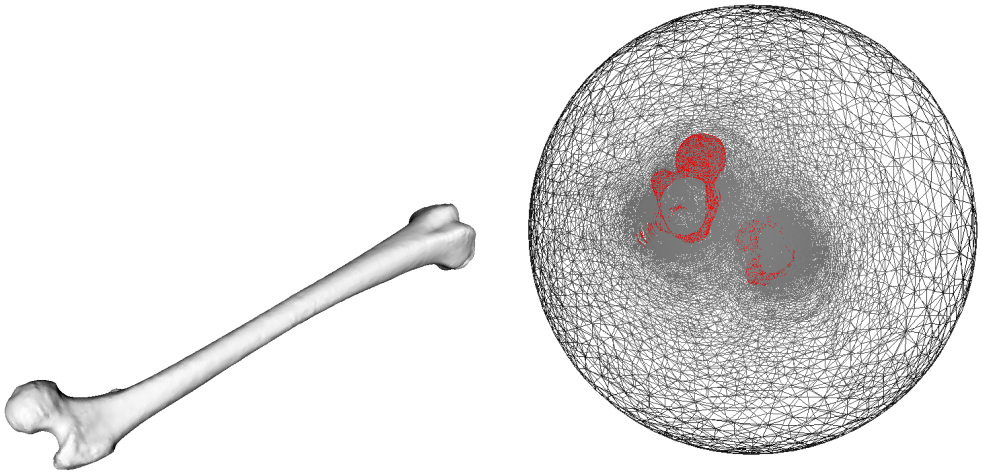


Figure 5.8: Result of non-rigid ICP alignment with all mesh points used as feature points. Self intersections visible in vertical line of central part.



(a) Femur bone (b) Spherical projection

Figure 5.9: Original femur bone model with 42 501 vertices and its spherical projection with overlap highlighted in red.

Relaxation method comparison

Various methods of spherical parametrisation were discussed in theoretical chapters 3 and 4. First one was Alexa's relaxation scheme described in 3.2 based on [1] (noted as *Ale00*). The similar relaxation scheme was described in 3.5 based on [27] (noted as *Zhu09*). I have tested them with both isolated iteration and in-place modification where results of current iteration can affect later processed parametric vertices in current iteration.

I have also tested the implementation of Parus (more in B.1.4) which I used for reference during implementation. However I was able to get identical results after debugging phase and therefore results of these measurements are fully represented by my C++ re-implementation.

To add some independent solution, I also tested parametrisation method described in [2] which I was able to obtain implemented (noted as *Ath11*). It introduces performance improvements and most importantly *OpenCL* implementation on GPU. Therefore the time results are not comparable with rest of my own CPU based implementations.

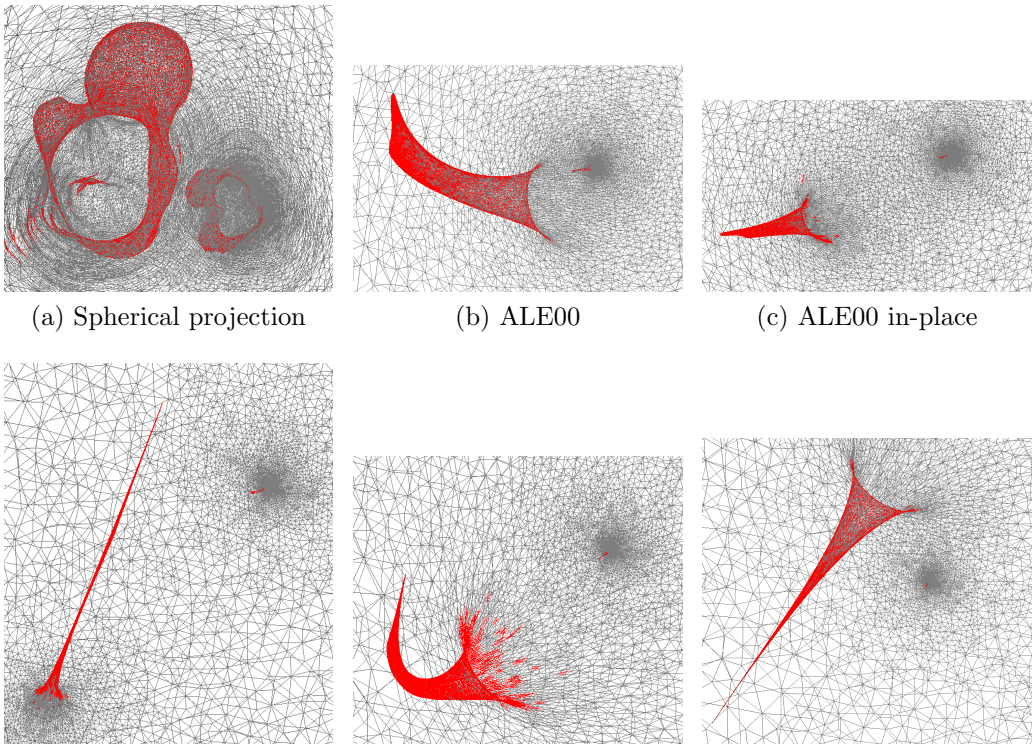
I have provided all my implementations with fully sufficient number of relaxation iterations set to 10 000. Changes with higher number of iterations are minimal. The goal was not to find fastest solution. The quality of parametrisation was main criteria instead. It was estimated using error function 4.15 based on area of flipped triangles. Results are summarised in Table 5.10.

Method	Time [ms]	Error [-]
Ale00	71 980	0,00170
Ale00 in-place	71 045	0,00063
Zhu09	18 952	0,00072
Zhu09 in-place	18 483	0,00067
Ath11	4 383	0,00070

Figure 5.10: Comparison of various parametrisation methods applied on single Femur bone mesh with 42 502 vertices.

As expected, no method was able to achieve perfect parametrisation without any overlapping triangles. This is weakness of relaxation method because the mesh is genus 0 and not that far from being star shaped. Hence perfect parametrisation is achievable for sure. The article [22] shows that this behaviour is common for iterative Gauss-Seidel based relaxation method as they become unstable after critical amount of iterations.

The differences in error size are not that important in practise as they are all wrong and will cause errors in morphing phase. Figure 5.11 shows details of the problematic regions Femur bone heads.



(a) Spherical projection (b) ALE00 (c) ALE00 in-place

(d) ZHU09 (e) ZHU09 in-place (f) ATH11

Figure 5.11: Comparison of mesh quality from various parametrisation methods. Detail look to mesh of Femur bone head where overlap regions exist (red).

The overlap regions emerge when group of vertices degenerates to single point. This then prevents neighbourhood shift to have significant effect and the relaxation slows down.

I decided to use *Ale00* as base for improvements because it generates more consistent overlap regions than *Zhu09*. The worse performance of *Ale00* is caused by algebraically more complicated relaxation formula. *Ath11* was not considered as choice because it is more complicated, harder to modify and does not seem to provide other benefits than the performance given by GPU implementation.

Influence of iteration count

I have further experimented with Alexa's relaxation schema with in-place modification. I have measured the residual error dependence on iteration count.

Table 5.12 shows that after fast initial phase, the relaxation process slows down and perfect state is never achieved. Adding even more iterations additionally causes some meshes to collapse. This is explained in [22] as attribute of Gauss-Seidel based methods. Usage of more than 10 000 iterations is therefore both ineffective and dangerous.

# of iterations	Error [-]
0	1,339E+06
100	0,147
200	0,130
300	0,115
400	0,102
500	0,088
600	0,076
700	0,064
800	0,054
900	0,046
1000	0,038

# of iterations	Error [-]
0	1,339E+06
1000	0,03849
2000	0,00866
3000	0,00365
4000	0,00225
5000	0,00160
6000	0,00122
7000	0,00098
8000	0,00082
9000	0,00071
10000	0,00063

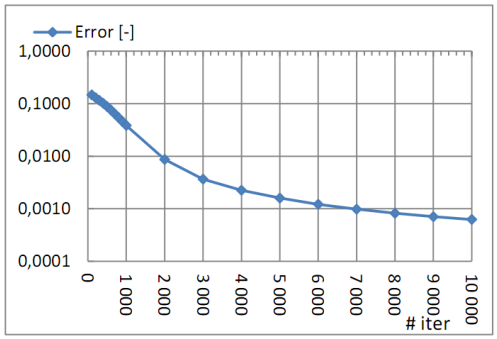


Figure 5.12: Influence of iteration count on residual error of Alexa's spherical parametrisation [1] measured by flipped triangle surface error function.

Influence of step size

I have also made measurements of influence of additional multiplication coefficient added to position change in relaxation of each vertex in equation 4.18.

Values under 1 makes the process slower while values above 1 use extrapolation to achieve larger correction shifts.

I have used 1 000 iterations only to make differences more significant.

Results in Table 5.13 seem to recommend the coefficient of 1.75 as ideal value. However results for coefficient 3.0 make warning that this improvement might be very unstable. This is then proven when coefficient 1.75 is applied with 10 000 iterations. It immediately fails to provide good parametrisation and exits with residual error of 0.02496 which is clearly more than with default coefficient 1.00 in previous Section which achieved residual error 0.00062. Optimal length of step may be also influenced by other parameters including size of mesh triangles and uniformity of their density distribution. I therefore do not recommend adding any additional coefficient for general usage without previous adaptation to specific input meshes.

Step size coefficient	Error [-]
0,25	0,13822
0,50	0,10858
0,75	0,07227
1,00	0,03849
1,25	0,01524
1,50	0,00402
1,75	0,00039
2,00	0,00817
3,00	1,34E+06

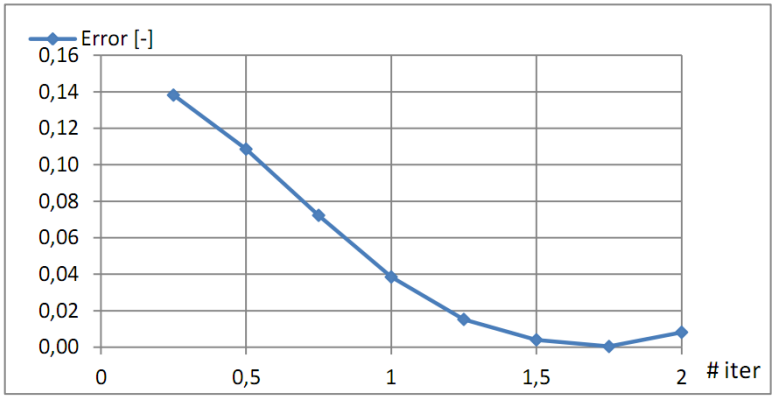


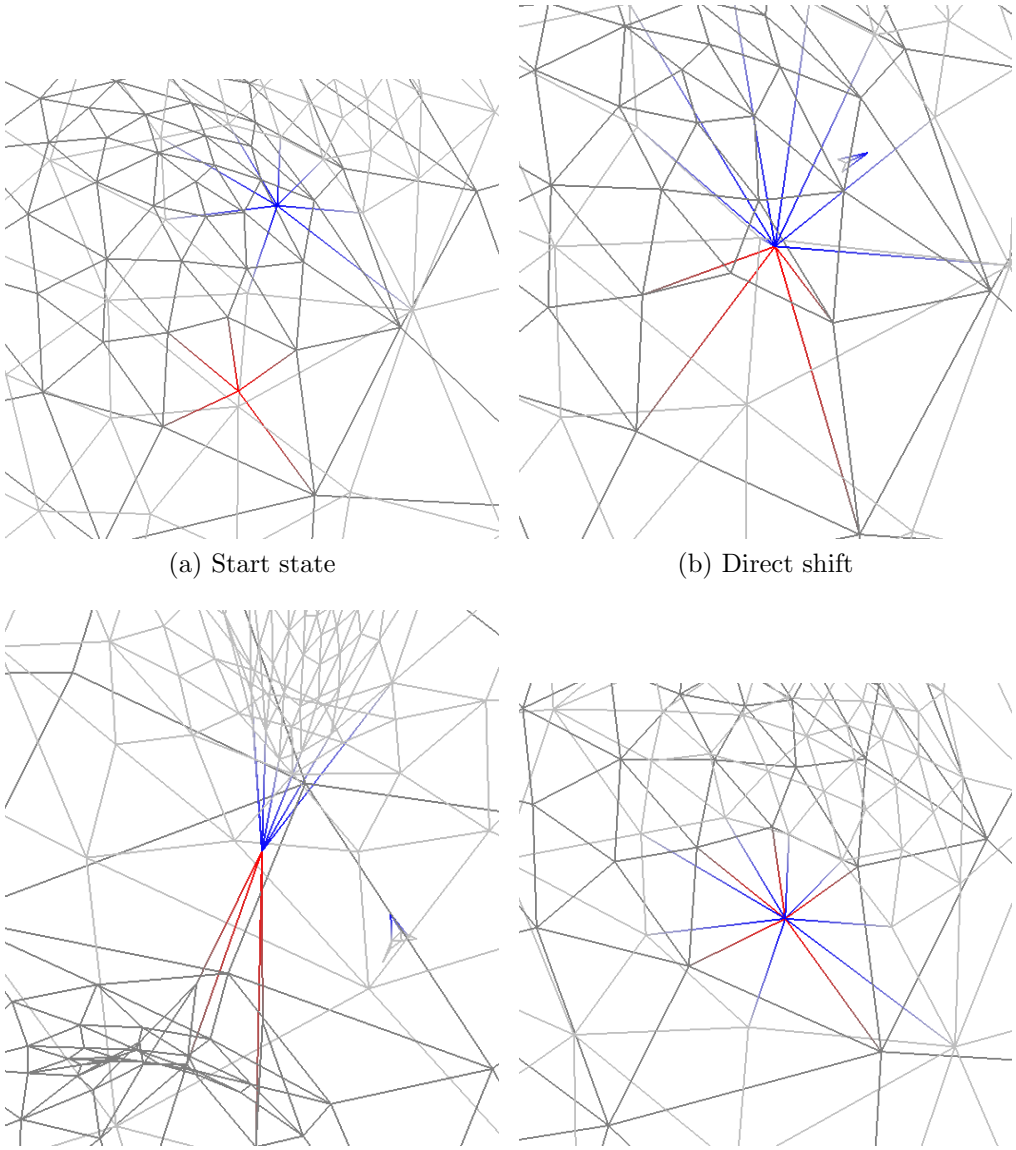
Figure 5.13: Influence of additional coefficient in relaxation step on residual error of Alexa's spherical parametrisation [1] measured by flipped triangle surface error function.

5.4.2 Spherical parametrisation adjustment distribution

Figure 5.14a presents alignment error in parametrisation of two meshes of reduced Femur bone models with 300 vertices each for better orientation in image. The red and blue vertices are chosen as matching feature points and should lie on the same place.

Zhu and Pang [27] suggest moving both feature points to normalised average position which causes overlaps in the mesh (Figure 5.14b). They use relaxation schema to fix this problem with one modification - feature vertices are locked and not relaxed at all. However Figure 5.14c shows that even 10 000 iterations of Alexa are not able to fix the problem. Furthermore it propagates the error to the rest of the mesh and even larger overlaps emerge.

My combined interpolation based shift and relaxation schema, which was de-



(a) Start state

(b) Direct shift

(c) Direct shift + relaxation

(d) Interpolated shift + relaxation

Figure 5.14: Alignment of pair of feature points (red and blue) on spherical parametrisations of two meshes reduced to 300 vertices (dark and light grey).

scribed in detail in Chapter 4.5.4, is able to provide aligned and valid parametrisation with only tens of iterations per shift (Figure 5.14d).

However the situation changes with bigger meshes where overlaps start to remain unfixed in dense areas of the parametric mesh (see Figure 5.15). I use smaller subshifts to reduce the problem, but it still persists. This problem is yet unresolved.

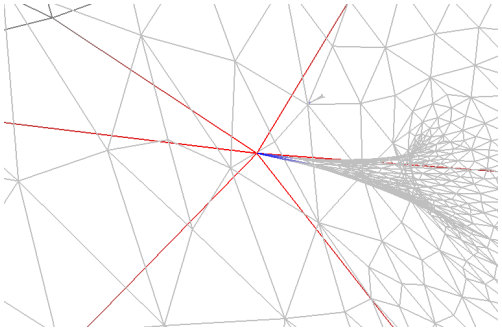


Figure 5.15: Problem with shift and relaxation of feature point alignment in dense areas of high-polygonal meshes.

5.4.3 Cascade spherical parametrisation

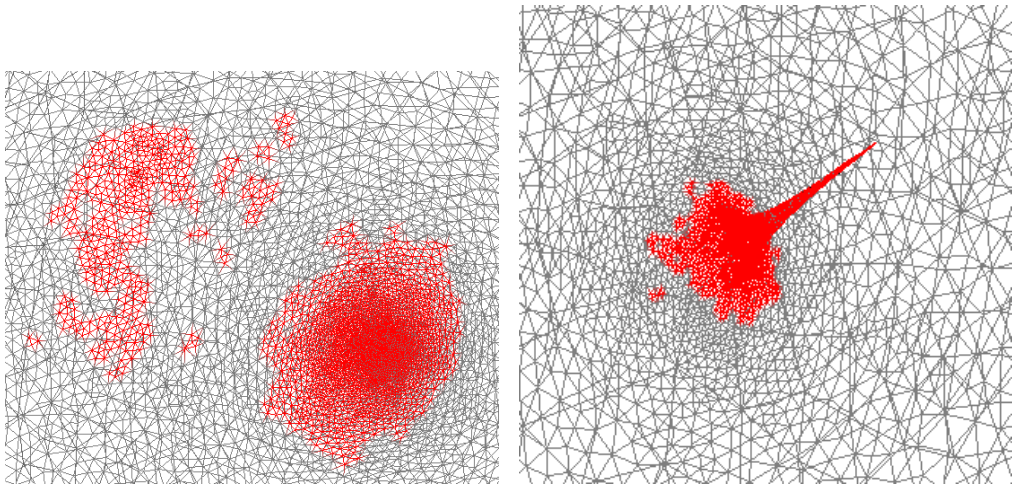
Section 5.4.1 shows that some meshes, especially those with higher number of vertices, tend to produce invalid parametrisation with overlapping triangles. To fix this, I have designed cascade schema exploiting the observation that low resolution meshes usually perform much better.

I have used model of Femur bone with 10 000 vertices to compare cascade approach with direct parametrisation. The cascade solution used coarse meshes of size 4 800, 2 400, 1 200, 600 and 300 to find parametrisations on lower levels and propagate it back to higher.

Direct comparison in Figure 5.16 show that not even one approach is successful in obtaining valid parametrisation.

Even worse, cascade mechanism leaves residual error 0.00016 which is even higher than 0.0000356 with direct approach. Adding execution time penalty of 95 seconds compared to 34 seconds and large memory consumption due to MVC matrices storage, cascade schema shows to be wrong way of fixing the problem.

The reason for failure lies in the last steps of up propagation where parametrisation of large coarse mesh is interpolated to the original size mesh. The resulting overlaps are unfortunately still too large and relaxation mechanism which is very unreliable for large meshes fails.



(a) Direct parametrisation (b) Cascade parametrisation

Figure 5.16: Comparison of spherical parametrisation output details in region of Femur bone head on model reduced to 10 000 vertices.

The conclusion is that direct approach is preferred where applicable.

5.5 Morphing

5.5.1 Barycentric coordinates on sphere surface

I have made experiment to find out, how big error can cause calculation of barycentric coordinates using planar method instead of spherical. I have used two average sized meshes with 2 502 vertices where both small and large triangles exists in parametrisation. I have taken all parametric vertices of first mesh and found their barycentric coordinates on other mesh. I then used these coordinates to reconstruct absolute coordinates of those points and measured distance from their template. This is equivalent to morphing of parametric domains itself.

In optimal case, zero error should be obtained. The Table 5.17 summarises results.

It is obvious that the error is much smaller for spherical coordinates. The difference is so large that it would surely be observable in result of morphing. Even bigger problem is that for some points no triangle can be found such that perpendicular projection falls into its area and its distance to the plane is within limit.

Method	Failed count	Failed ratio	Total error	Average error
Planar	114	5%	112,34800	0,04490
Spherical	0	0%	0,48358	0,00019

Figure 5.17: Comparison of calculation of barycentric coordinates on spherical domain using spherical approach and planar simplification. Mesh with 2 502 vertices used.

5.5.2 Spherical and direct mesh domain comparison

Although I was unable to fully fix all problems with spherical parametrisation discussed in 5.4.1, I decided to use meshes that have minimal smallest parametrisation problems and compare results of direct domain morphing with spherical parametric domain.

I have used two smaller Femur bone models and two Iliacus muscle models as their shape is relatively good for spherical parametrisation.

Figure 5.18 shows that even for such simple model, cascade parametrisation is not able to produce valid results. The shape is heavily distorted and the situation is even worse with cascade schema, that was already been proven to be inefficient.

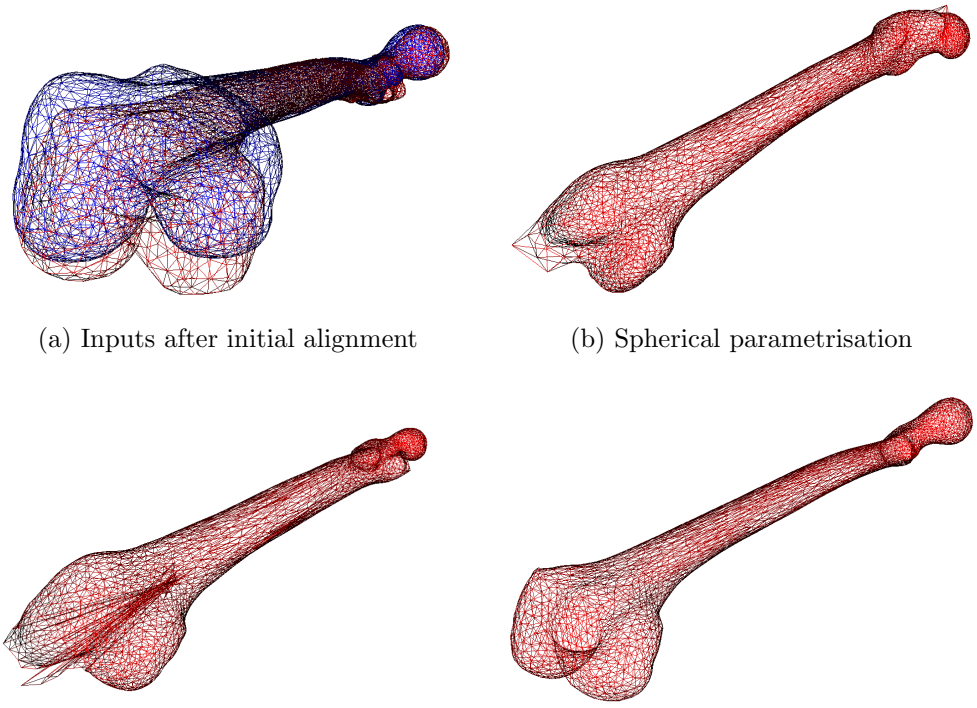
The reason for peak at bone head can easily be found in Figure 5.19. The result of spherical variant is significantly compressed and the parts that stays closer to original position then creates such peak. This is result of long interpolation paths discussed in chapter of 4.5.4.

The Iliacus muscle results seem better for spherical parametrisation if solid surface is used for rendering, but wireframe render shows inner mesh problems caused by misaligned parametrisations (Figure 5.20). Even worse results come from cascade schema.

This all proves that valid, not overlapping parametrisation is vital for spherical domain methods and if this is not possible to guarantee, especially when mutual alignment of domains is added to the case, results are very poor.

The results of direct mesh morphing are on the other side satisfying and therefore I will use it exclusively in rest of experiments.

Because direct morphing effectively skips one step of algorithm (parametrisation), it is not a surprise, that measured execution times are on its side as well (see Table in Figure 5.21).



(a) Inputs after initial alignment

(b) Spherical parametrisation

(c) Cascade spher. parametrisation

(d) Direct morphing

Figure 5.18: Comparison of morphing of two Femur meshes (a) using various parametric domains (b, c, d).

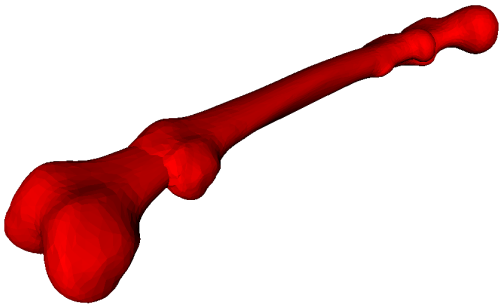


Figure 5.19: Overlay of Femur bone morphing outputs from spherical parametrisation method (smaller) and direct morphing (larger).

5.6 Overall

5.6.1 Final results

In this chapter, I present and discuss outputs of the method for various input meshes. All experiments were done using direct morphing version of the algorithm.

Manifold inputs

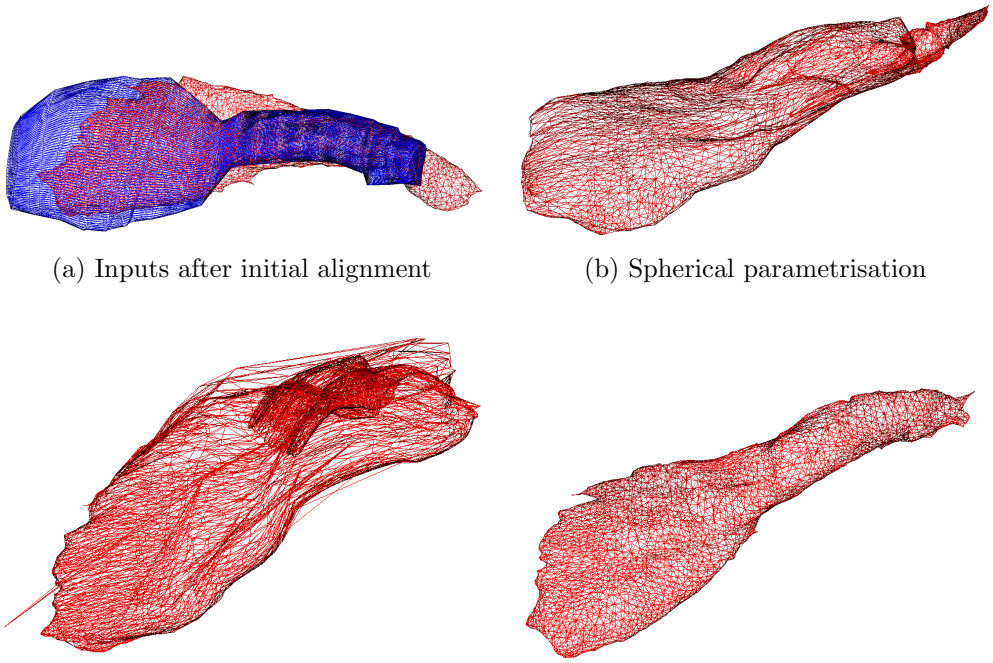
Morphing of manifold *Femur* bones was easy as the non-rigid *ICP* was able to produce very good alignment of parametric domains. Therefore the resulting mesh has smooth shape without any visible problems (see Figure 5.22).

Although the alignment of much more varying meshes in *Iliacus* set is looser, the result is still very good (see Figure D.1) thanks to normal direction constraint in point-to-cell pairing mechanism of morphing.

Worse results were gained for thin and irregularly shaped models of *Sartorius* muscle. When two more robust and less peaked meshes in Figure D.2 were used, the results seemed good. However if all three meshes including the large and jagged one was put on input, the resulting mesh seems to have problematic regions on one pole. It maintains manifold edges but some parts are close self intersections of surface (see Figure D.3).

The jagged shape in combination with high vertex count seems to cause problem in local alignment as applied number of *ICP* regions (up to 500^3) does not guar-

[3]Higher numbers of regions did not seem to improve alignment and caused higher chance of



(a) Inputs after initial alignment

(b) Spherical parametrisation

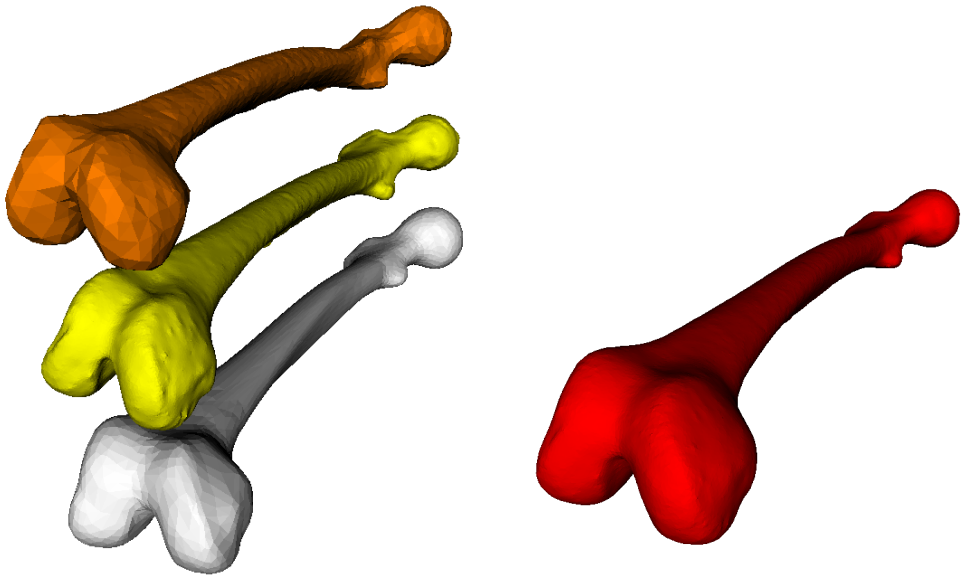
(c) Cascade spher. parametrisation

(d) Direct morphing

Figure 5.20: Comparison of morphing of two Iliacus muscle meshes (a) using various parametric domains (b, c, d).

Data set	Vertices	Method time [ms]		
		Direct	Spherical	Spherical cascade
Femur	2 x 2502	18 851	62 158	119 561
Iliacus	3248 + 3978	40 927	111 456	280 985

Figure 5.21: Execution times of complete algorithm using various domain for morphing.



(a) Partially aligned inputs (b) Output of morphing

Figure 5.22: Output of morphing of three manifold Femur bone models.

antee precise enough alignment in such case. Laplace smoothing of such input might help resolve this issue, but was not tested.

Non-manifold or combined inputs

Non-manifold *Sartorius* muscle model featured only 2 non-manifold edges, but its thin shape and jagged outlier was potential source of problems. However using robust model of the same muscle with almost none features, morphing was done giving plausible output (see Figure D.4). The output keeps all small features of the non-manifold model but consists of only manifold edges and vertices. Isolated components were removed as well.

I used two artificially created damaged meshes of *Femur* bone to test morphing of two non-manifold meshes. The initial result in Figure 5.23a shows noticeable

self-intersections in deformation phase of non-rigid *ICP*.

displacements in areas where holes were filled and reliability weights were therefore zero. This caused usage of only one mesh and therefore sudden jump in the surface.

(a) Refined input meshes with weights

(b) Refined input meshes without weights

(c) Non-rigidly aligned meshes with weights

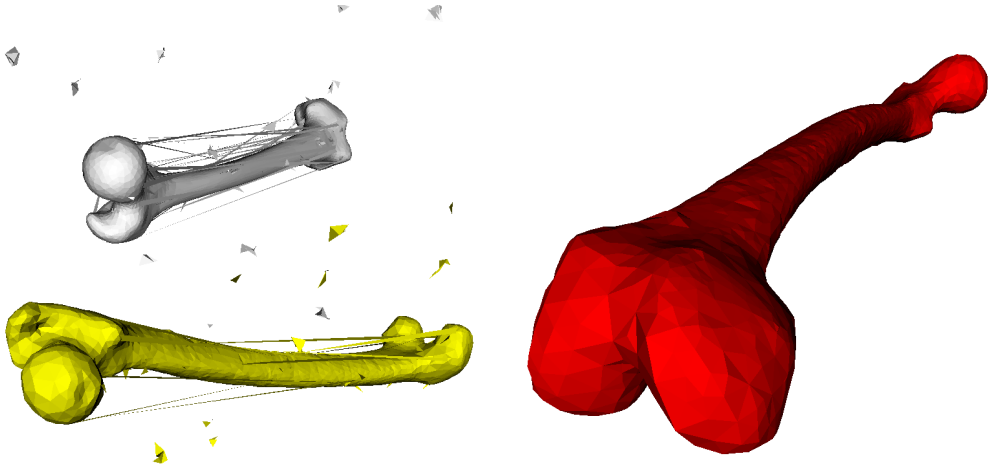
Figure 5.23: Various approaches for multi-morphing of damaged meshes.

I propose two possible solutions. First, we could just abandon the idea of reliability weights and always interpolate between all input surfaces. The output of such idea in Figure 5.23b presents that this produces smooth and bump free surface. However it might be risky if there was larger a hole in some area of any input. It might then cause the final mesh to be unnecessarily flattened in such area unless you fill holes using a better method - see Section 4.2.4.

Therefore I suggest different solution that keeps reliability weights in action, but reduces the jumps between them to minimum. Figure 5.23c shows that the surface is much smoother if results of non-rigid *ICP* alignment were used as inputs.

It might look like that identical shapes are then being morphed, but that is not usually true as the non-rigid alignment rarely results into perfect fitting like with the Femur bone.

The final result of morphing of fully aligned meshes with reliability weights can be seen in Figure 5.24 together with inputs used.



(a) Inputs (b) Output of morphing

Figure 5.24: Output of morphing (red) two intentionally non-manifold models of Femur bone (yellow, white).

5.6.2 Timings

I measured execution times for individual steps of direct morphing variant of the algorithm.

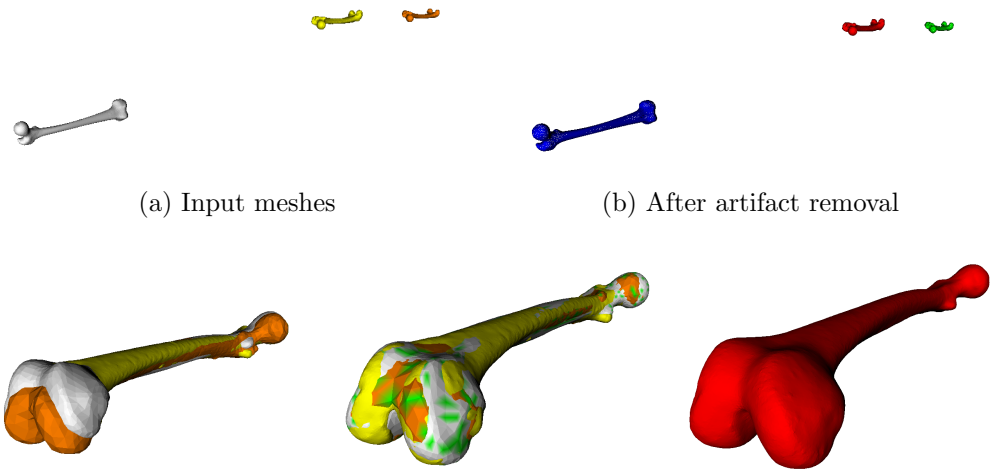
The output of individual steps for three manifold femur bone data set is visualised on Figures 5.25. Figure 5.25a shows input meshes in their start positions. Figure 5.25b shows result of step 1 - artifact fix. It removes non-manifolds, isolated components and fills holes. Step 2 is initial global space alignment using *PCA* (Figure 5.25c). The algorithm continues with step 3 and non-rigid *ICP* alignment using my region selection based on k-neighbourhood (Figure 5.25d). Finally, all meshes are morphed using direct morphing domain in step 4 (Figure 5.25e).

The results of time measurements are summarised in Table 5.26.

It can easily be seen that about 70% of time is spent in rigid mesh alignment step. It is caused mainly by relatively large number (usually the maximum of 500) of deformation regions resolved using *ICP*. It could probably be significantly accelerated if spatial subdivision structure was used for finding the nearest point. This is however valid assumption even for morphing and initial alignment step as they all perform vertex to vertex search operations.

Absolute values of times are by far not interactive but that was not the goal. They are low enough so that complete muscles set of human body model can be processed in few hours.

If shorter execution times were required, more advanced algorithm of space search improvement would again be most vital change.



(a) Input meshes (b) After artifact removal

(c) After init. *PCA* alignment (d) After non-rigid ICP (e) Output of direct morphing

Figure 5.25: Outputs of individual main steps of the complete morphing algorithm for three Femur bones.

Data set	Vertex counts		S1: Fix artifacts		S2: Initial alignent		S3: Non-rigid ICP		S4: Morphing		Total
	Input	Output	Time [ms]	Ratio	Time [ms]	Ratio	Time [ms]	Ratio	Time [ms]	Ratio	Time [ms]
Femur	2x2502 + 42502	42 502	665	0%	15 419	3%	428 338	73%	141 290	24%	585 712
Iliacus	3248 + 3978	3 978	195	0%	4 540	9%	35 029	73%	8 408	17%	48 172
Sartorius	2390 + 4956	4 956	115	0%	3 255	9%	23 914	68%	7 777	22%	35 061
Sartorius	2390 + 4956 + 9004	9 004	231	0%	7 524	4%	138 431	73%	43 485	23%	189 671
Femur NM	2698 + 2629	2 500	153	1%	3 192	16%	12 690	63%	4 160	21%	20 195
Sartorius NM	9001 + 2390	8 986	287	0%	4 729	6%	62 354	77%	13 879	17%	81 249

Figure 5.26: Execution times of individual steps of the direct morphing algorithm for various inputs. NM denotes non-manifold inputs.

Then parallelisation could also be easily applied as most of operations are run on individual meshes, vertices or triangles, so they independent on each other. There might only be need for change of some data structures and their access methods. For instance, *std* library in *C++* or direct getters in *VTK* like parameterless overload of `vtkPolyData::GetPoint()` are not thread safe.

5.6.3 Application to human body framework

I have used non-manifold mesh sets to test influence of morphing on quality and precision of deformation using volume preserving algorithm from [13].

This experiment was relatively complicated to realise as I was no longer able to build application from my bachelor work from sources. This was caused by loss

of appropriate versions of referenced libraries as the framework I worked with does not compile with current versions.

Therefore I extracted the algorithm and temporarily integrated it into my application. It was removed again after experiments so that it does not collide with the new source code. It would also make no benefit for the user without a convenient way of defining action lines for skeletons. Such extension would be beyond this work's scope.

This also means that the skeleton action lines are manually inserted approximation of real muscle bounds. This however does not influence the informative value of test.

Simple *vtkQuadricDecimation* together with enlargement offset was used to create coarse mesh in the deformation algorithm before the new decimation filter *vtkProgressiveHull* (see Section B.1.3) was implemented. This was the case in the bachelor thesis as well. I have experimented with both solutions this time as they give very different results.

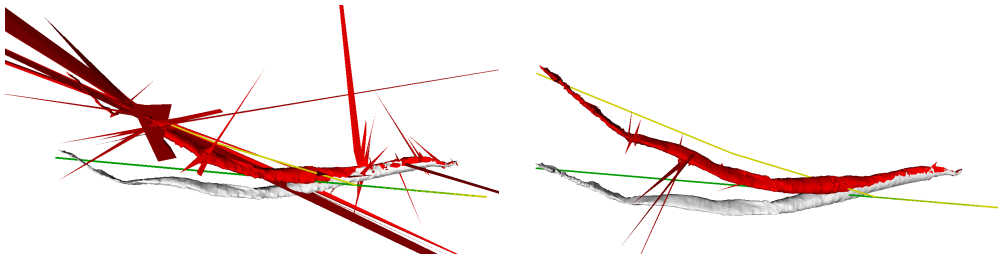
Sartorius muscle

I have compared results of deformation for original non-manifold Sartorius muscle and output of its morphing with manifold model from Chapter 5.6.1 (see Figure D.4 for both).

With older *vtkQuadricDecimation* decimation filter, the deformation of non-manifold mesh resulted into heavy distortion (see Figure 5.27a).

The same method applied on morphed mesh did not achieve perfect result but significantly reduced the distortion (see Figure 5.27b).

Although both outputs featured large problems, the volume preservation coefficient defined as ratio of output and input volume $c = \frac{V_{out}}{V_{in}}$ was maintained 0.974571 and 1.00196 which itself would be acceptable.

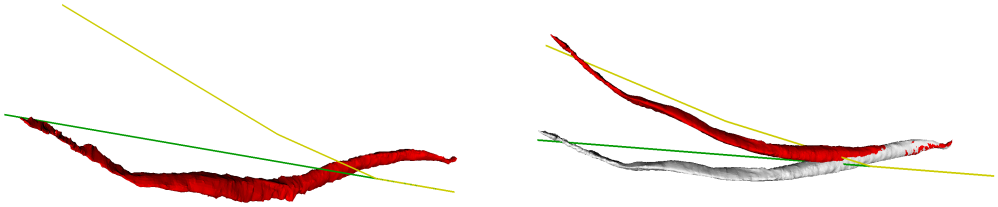


(a) Deformation of original mesh (b) Deformation of multi-morphed mesh

Figure 5.27: Deformation of original non-manifold and morphed manifold Sartorius muscle using old version of deformation filter from [13].

When newer *vtkProgressiveHull* decimation filter was activated in the deformation filter, the deformation of non-manifold mesh failed. I have made short investigation of the reason. I have found out, that the matrix of linear equation is poorly conditioned and its inversion fails. This then caused deformation process to stop (see Figure 5.28a).

The deformation of the morphed mesh on the other hand produced valid result without any artifacts and with perfect volume preservation coefficient of 1.00031 (see Figure 5.28b).



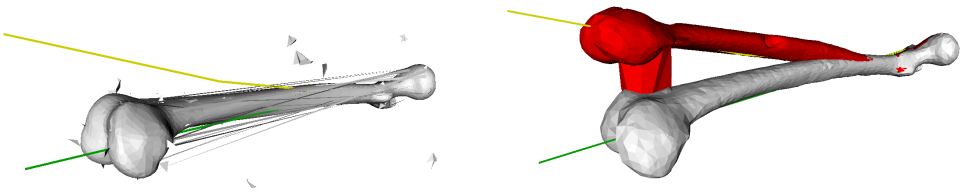
(a) Deformation of original mesh (b) Deformation of multi-morphed mesh

Figure 5.28: Deformation of non-manifold and morphed sartorius muscle using new version of deformation filter from [13].

Femur bone

Here I used both artificially created non-manifold damaged models of femur bone and compared results of deformation of one of them with result of deformation of their morphing output as described in Chapter 5.6.1 (see Figure 5.24 for both).

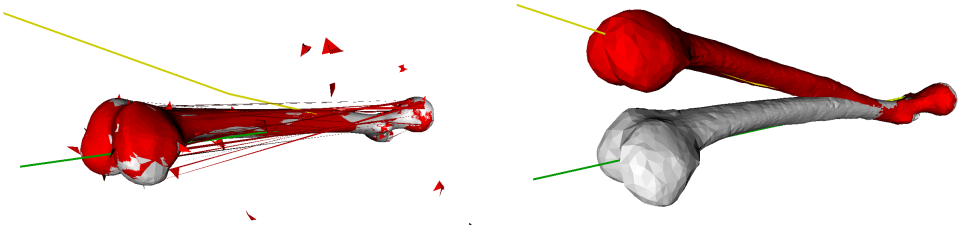
Old modification of deformation filter failed on non-manifold input (see Figure 5.29a). The same filter applied on output of morphing resulted in large artifacts on output caused by poor coarse mesh (see Figure 5.29b). The volume preservation coefficient 1.00614 does not balance such failure.



(a) Deformation of original mesh (b) Deformation of multi-morphed mesh

Figure 5.29: Deformation of non-manifold and morphed femur bone using old version of deformation filter from [13].

The new modification of new filter behaves much better. Although it once again fails on near-to-singular matrix inversion for non-manifold input (see Figure 5.30a), it provides valid results without noticeable problems for morphed mesh (see Figure 5.30b). Same applies on the volume preservation coefficient where perfect value of 1.00065 was achieved.



(a) Deformation of original mesh (b) Deformation of multi-morphed mesh

Figure 5.30: Deformation of non-manifold and morphed femur bone using new version of deformation filter from [13].

Summary

The morphing itself does not guarantee that output mesh will behave well in deformation filter [13] but it seems to at least reduce the extent of problems. The results are much better and more reliable with new decimation filter *vtkProgressiveHull*, which was not tested in the original work [13].

5.6.4 Final method

After all presented experiments, the final properties of multi-morphing method were chosen. The direct morphing schema outperformed the spherical domain alternative, therefore the schema in Figure 5.31 is recommend for general usage. This means that the spherical parametrisation step was completely removed. If you still wanted to use the spherical domain version, standard parametrisation with in-place Alexa 2000 relaxation schema without the cascade extension is recommended.

The individual steps are then recommended to be used as follows:

1. *Artifact removal* - use as described in Section 4.2. No ambiguous alternatives were provided.

2. *Initial alignment* - use as *PCA* and progressive coarse mesh for acceleration

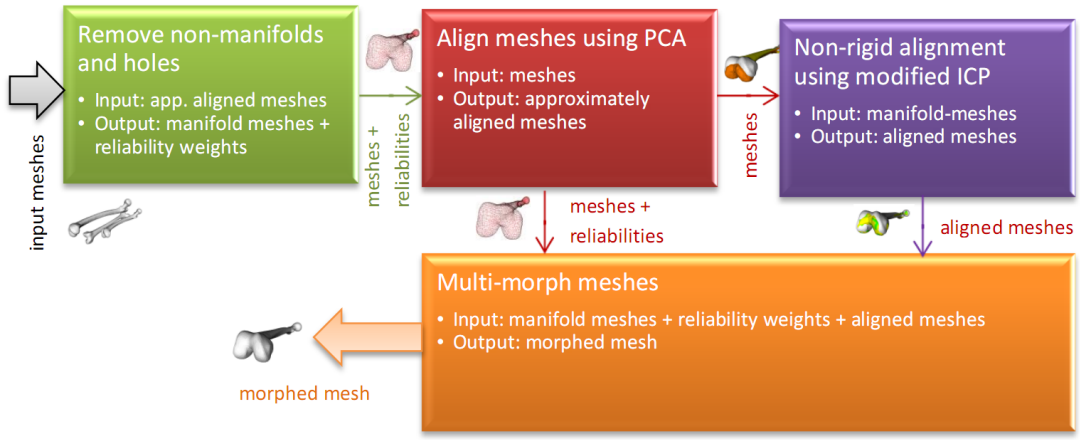


Figure 5.31: Flow diagram for direct on-mesh morphing version of algorithm.

3. *Non-rigid alignment* - use as rigid whole mesh *ICP* first, then continue with non-rigid *ICP* method on regions generated by 3-neighbourhood. Use up to 500% regions, 10% of mesh vertices typically.

4. *Morphing* - use direct morphing on non-rigidly aligned meshes. Apply reliability weights from step 1.

Details of individual steps and their mentioned variants are detail discussed in Chapter 4.

Chapter 6

Conclusion

I have introduced one of problems being solved in the *VPHOP* project and stated the role of this thesis in the introduction chapter. I have then provided wide description of various registration techniques including those suitable for non-rigid objects. I have also summarised morphing approaches with both general algorithm and specific examples. I have then evaluated the suitability of individual solutions for this thesis and proposed an automatic multi-morphing method for non-manifold genus 0 meshes. I have implemented several variations and focused mainly on spherical parametrisation and non-rigid *ICP* based registration parts where I introduced some modifications and provided a comparison of their performance.

Some of the ideas did not prove to be effective, such as cascade spherical parametrisation. The spherical parametrisation itself then had to be abandoned as it was unable to provide valid parametrisation no matter what modification and public available solution I tested.

Finally I created a specification for implementation that I was able to successfully test in both partial and complex experiments with input meshes consisting from both manifold and non-manifold real muscle meshes. The method behaves well for smooth meshes, but it can produce self-intersections in jagged regions. This could potentially be fixed in post-processing.

I have compared benefit of method to deformation filter on human body framework and it showed noticeable improvement in stability and quality of output.

I have also provided asymptotic time analysis of current implementation and I suggest acceleration using spatial subdivision data structures such as kd-trees for space search operations. Another performance improvement could easily be gained using parallelisation by *OpenMP* library that I worked with in previous work.

The main goal for future is to make the search of parametric domain more robust to shape change. Non-rigid *ICP* method used is only able to make good alignment for close surfaces of similar shape. Especially my adaptation based on k-neighbourhood would provide unstable results with larger changes. Therefore in current state, this method cannot morph between completely different objects, such as cow and fish, which was presented in some other works like for example

[27]. They however used spherical parametrisation and I was unable to reproduce their result. After consultation with Ing. Jindřich Parus Ph.D., I suspect those algorithms not to be complete or not to be applicable to so general data as they claim.

However for the aim of this work, muscle models are expected to be reasonably similar so the method seems to be usable. By providing exhaustive theoretical introduction and description of existing methods, suggesting a new method based on their combination, its implementation and testing from various aspects, concluded with evaluation of gained results, I believe the assignments of this work were fulfilled.

Abbreviations

AMS ... Affine Morphing Space

BFS ... Breadth First Search - general graph search algorithm

DFS ... Depth First Search - general graph search algorithm

DLL ... Dynamic-link library

GRB ... Global Reference Body - global reference system for medical data of *VPH*

GUI ... Graphical User Interface

ICP ... Iterative Closest Point - basic method for rigid mesh registration

MVS ... Morphing Vector Space

VPH ... Virtual Physiological Human - a framework of methods and technologies that will make it possible to describe human physiology and pathology in a complete and integrated way

VTK ... The Visualization Toolkit – graphical library pro visualisation and manipulation of various data

Bibliography

[1] M. Alexa. Merging polyhedral shapes with scattered features. *The Visual Computer*, 16:26–37, 2000. ISSN 0178-2789. URL http://dx.doi.org/10.1007/PL00007211. 10.1007/PL00007211.

[2] T. Athanasiadis and I. Fudos. Smi 2011: Full paper: Parallel computation of spherical parameterizations for mesh analysis. *Comput. Graph.*, 35:569–579, June 2011. ISSN 0097-8493. doi: http://dx.doi.org/10.1016/j.cag.2011.03.022. URL http://dx.doi.org/10.1016/j.cag.2011.03.022.

[3] G. Barequet and M. Sharir. Filling gaps in the boundary of a polyhedron. *Computer Aided Geometric Design*, 12(2):207 – 229, 1995. ISSN 0167-8396. doi: 10.1016/0167-8396(94)00011-G. URL http://www.sciencedirect.com/science/article/pii/016783969400011G.

[4] P. J. Besl and N. D. McKay. A method for registration of 3-d shapes. *IEEE Trans. Pattern Anal. Mach. Intell.*, 14:239–256, February 1992. ISSN 0162-8828. doi: 10.1109/34.121791. URL http://dl.acm.org/citation.cfm?id=132013.132022.

[5] B. Brown and S. Rusinkiewicz. Global non-rigid alignment of 3-D scans. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 26(3), August 2007.

[6] D. Cholt and J. Kohout. Progressive hulls: Application on biomedical data. In *Proceedings of CESCG 2012: The 16th Central European Seminar on Computer Graphics*, 2012.

[7] Nokia Corporation. Qt documentation, 2011. URL http://www.vtk.org/doc/release/4.0/html/index.html. Checked 2012-2-1.

[8] Kronos Group. Opengl 2.1 reference pages, 2012. URL http://www.opengl.org/sdk/docs/man/. Checked 2012-4-24.

[9] W. Guan. Visualization toolkit (vtk), part i, 2009. URL www.rhpcs.mcmaster.ca/~guanw/course/vtk1.pdf. Checked 2011-3-9.

[10] D. van Heesch. Vtk documentation, 2001. URL http://www.vtk.org/doc/release/5.8/html/. Checked 2012-2-21.

[11] Q. Huang, B. Adams, M. Wicke, and L. J. Guibas. Non-rigid registration under isometric deformations. *Computer Graphics Forum*, 27(5):1449–1457, 2008. ISSN 1467-8659. doi: 10.1111/j.1467-8659.2008.01285.x. URL http://dx.doi.org/10.1111/j.1467-8659.2008.01285.x.

[12] I. Jolliffe. Principal component analysis. Springerg, 2 edition, 2002. ISBN 0-387-95442-2.

[13] P. Kellnhofer. Deformation of surface models with volume preservation. 2009. URL https://portal.zcu.cz/wps/PA_StagPortletsJSR168/KvalifPraceDownloadServlet?typ=1&adipidno=37212. Czech language only. Checked 2012-01-08.

[14] P. Kellnhofer. Non-rigid transformations for musculoskeletal model. 2012. URL https://portal.zcu.cz/wps/PA_StagPortletsJSR168/KvalifPraceDownloadServlet?typ=4&adipidno=46323. Checked 2012-07-09.

[15] J. R. Kent, W. E. Carlson, and R. E. Parent. Shape transformation for polyhedral objects. *SIGGRAPH Comput. Graph.*, 26:47–54, July 1992. ISSN 0097-8930. doi: http://doi.acm.org/10.1145/142920.134007. URL http://doi.acm.org/10.1145/142920.134007.

[16] Kitware. Cmake - cross platform make, 2012. URL http://www.cmake.org/. Checked 2012-3-15.

[17] J. Kohout, P. Kellnhofer, and S. Martelli. Fast deformation for modelling of musculoskeletal system. In *Proceedings of the International Conference on Computer Graphics Theory and Applications: GRAPP 2012*, pages 16–25, Rome, February 2012.

[18] V. Kraevoy and A. Sheffer. Cross-parameterization and compatible remeshing of 3d models. *ACM Trans. Graph.*, 23:861–869, August 2004. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1015706.1015811. URL http://doi.acm.org/10.1145/1015706.1015811.

[19] P. Liepa. Filling holes in meshes. In *Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, SGP '03, pages 200–205, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. ISBN 1-58113-687-0. URL http://dl.acm.org/citation.cfm?id=882370.882397.

[20] J. Parus. Morphing of geometrical objects in boundary representation. 2009. URL http://portal.zcu.cz/wps/PA_StagPortletsJSR168/KvalifPraceDownloadServlet?typ=1&adipidno=23343. Checked 2012-01-08.

[21] S. Rusinkiewicz and M. Levoy. Efficient variants of the icp algorithm. In *INTERNATIONAL CONFERENCE ON 3-D DIGITAL IMAGING AND MODELING*, 2001.

[22] S. Saba, I. Yavneh, C. Gotsman, and A. Sheffer. Practical spherical embedding of manifold triangle meshes. In *Proceedings of the International Conference on Shape Modeling and Applications 2005*, pages 258–267, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2379-X. doi: 10.1109/SMI.2005.32. URL http://dl.acm.org/citation.cfm?id=1097876.1098477.

[23] W. Schroeder, K. Martin, and B. Lorensen. *The Visualisation Toolkit, Third Edition*. Kitware Inc., 2004.

[24] J. Tao, S. Scott, and D. W. Joe. Mean value coordinates for closed triangular meshes. *ACM Trans. Graph.*, 24(3):561–566, 2005. URL http://dblp.uni-trier.de/db/journals/tog/tog24.html#JuSW05.

[25] M. Viceconti and G. Clapworthy. *VPH-FET Research Roadmap - Advanced Technologies for the Future of the Virtual Physiological Human*. VPH-FET consortium, 2011.

[26] VPHOP. Osteoporotic virtual physiological human, 2009. URL http://vphop.eu. Checked 2011-4-25.

[27] Z.-J. Zhu and M.-Y. Pang. Morphing 3d mesh models based on spherical parameterization. In *Proceedings of the 2009 International Conference on Multimedia Information Networking and Security - Volume 01*, MINES '09, pages 309–313, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3843-3. doi: http://dx.doi.org/10.1109/MINES.2009.29. URL http://dx.doi.org/10.1109/MINES.2009.29.

Appendix A

User documentation

All following instructions are targeted to *MS Windows 7 Professional x64* operating system and *MS Visual Studio 2010 Ultimate* integrated development environment. However it should be applicable also to operating system *Windows XP 32-bit* or newer.

The libraries and most of the source code should compile on *GNU/Linux* as well but some small fragments use *Windows API* which makes compilation elsewhere impossible. See Chapter 5.1 for details and information about necessary changes. I also do not provide any makefile for alternative compilers.

The same applies to provided binary files that can run on *Windows XP 32-bit* or newer *Microsoft* operating system.

A.1 Build

There are several folders in root of attached *DVD*. Their description can also be found in the `readme.txt` file.

Folder *src* contains C++ source codes and solution for *MS Visual Studio 2010 Ultimate* in file *MeshRegister.sln*. When you open it, you can see two main projects.

To compile the *MeshRegister* project with algorithm implementation, paths to *VTK 5.8.0* or newer must be set in project properties. You can find source codes of *VTK* in *extra* folder in root of *DVD* or you can download it from *VTK* homepage [10]. The *VTK* itself has to be configured using *CMake* [16] and compiled first.

For compilation of the *MeshRegisterGUI* project with GUI, paths to the same *VTK 5.8.0* library and to *Qt 4.7.4* or newer library has to be set up. Again, you

can obtain the second one from *extra* folder on *DVD* or from internet [7]. You can also download extension for *Visual Studio* here to make work with framework easier.

Then you can compile both projects using *Build All* function of *MSVS*. You should not see any errors or warnings. The output file *MeshRegisterGUI.exe* will be created.

To run the application, copy all other *DLL* libraries and folder structure from *bin* folder on *DVD* to the same folder where built *MeshRegisterGUI.exe* resides. You will have to unpack them by yourself or use the installer which unpacks them automatically. Then you can run the application using this executable.

A.2 Installation and prerequisites

Open *bin* folder on attached *DVD*. Run the *setup.exe* and go through standard installation wizard. You will need about 25 MiB of free space on target hard drive.

Application requires *Microsoft Visual C++ 2010 Redistributable Package (x86)* to run. Installer should be able to detect it and download it from internet if necessary.

By default, folder *MeshRegisterGUI* will be created in *Start* menu with shortcut *MeshRegisterGUI* leading to installation directory.

Here you can find main executable *MeshRegisterGUI.exe*. Use it to start the application.

You also find here a few other folders. Folder *examples* contains some demo data for testing. Folder *cache* will be used for output caching. Folder *logs* will be filled by application logs. Folder *data* contains internal application data.

If you want to remove application, then close it and use standard *Windows* application manager *Programs and features* accessible though *Control panel*. Here you will find it as *MeshRegisterGUI*.

A.3 User manual

After application is started, you see the main window (Figure A.1).

It can be divided into three main zones.

The upper left panel shows rendering of input or output meshes. Use the tab controls under the panel to switch between them.

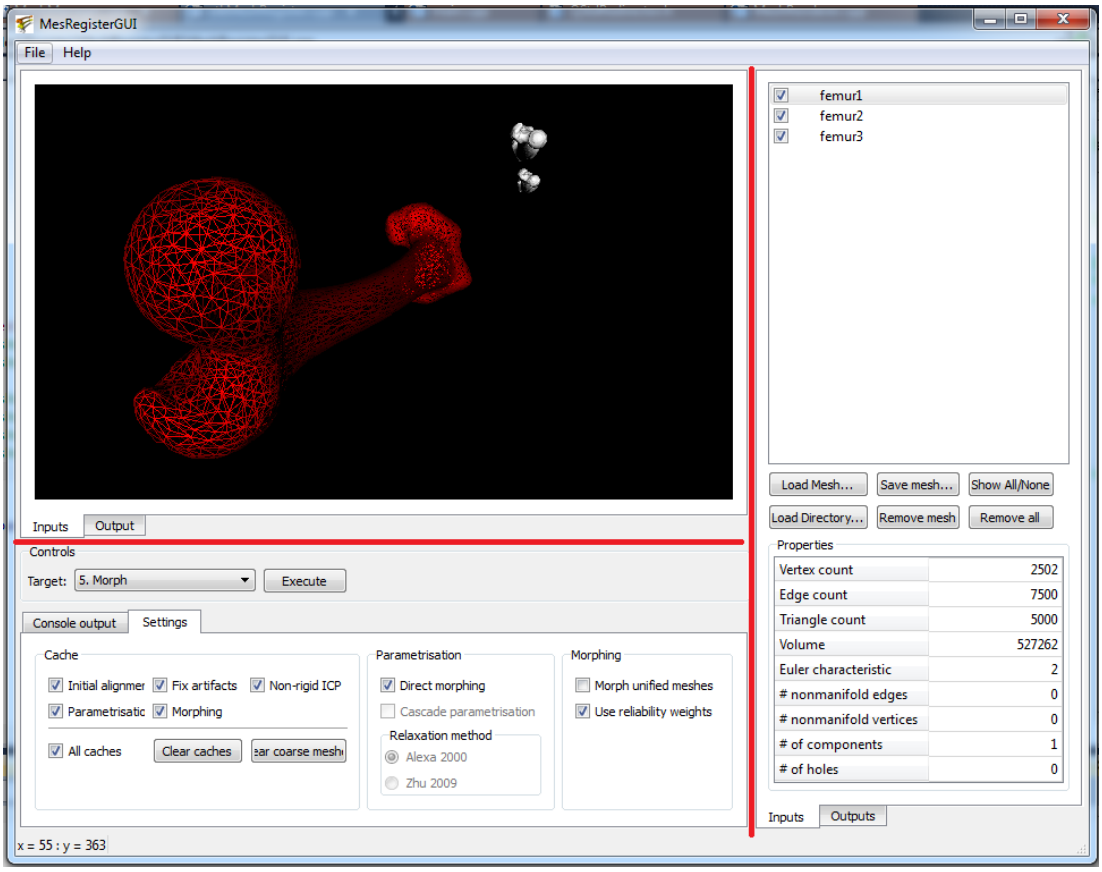


Figure A.1: Main window of the MeshRegisterGUI application with input tab selected. Red lines highlight the splitters for change of space ratios between panels.

The bottom left panel contains tabs with console output and morphing settings. It also contains the execution button in its upper part.

The right vertical panel then holds manger of both input and output meshes. It again contains tab switch between inputs and outputs which is synchronised with renderer panel, so you always see what you manage and vice-versa.

Drag the red highlighted zones on figure A.1 with your left mouse button to change the space ratio between those components.

A.3.1 Mesh management

After start with empty application, you will see empty input mesh manager (Figure A.1). You can use button *Load Mesh...* to add single mesh using file opening dialog. This will allow you to select any supported file which is *VTK* or *OBJ*. Support of *OBJ* may not be perfect and was tested with only limited number of exporters.

You can also use *Load Directory...* button to open all supported files in selected directory at once. It is very useful with example inputs prepared in folder *examples* in installation directory.

Loading the meshes may take same time. They are added to the upper list with checkboxes (Figure A.1). Initially no mesh is selected.

You can now click name of any mesh to see its parameters in the bottom table of the mesh manager (Figure A.1).

Then you may tick some meshes using checkboxes left from their names. This will add individual meshes to the renderer view. Those meshes will also be considered input for morphing.

The selected mesh with displayed statistics is shown red and wireframe in the left window. Others are white and solid (Figure A.1).

You can also remove single selected mesh using *Remove Mesh* button or all meshes in manager using *Remove All* button.

The button *Save Mesh...* opens a save file dialog and allows you to export single selected mesh into one of three supported formats:

- *VTK*
- *OBJ*
- *TRI*

It is more useful for output meshes but it is enabled in input manager as well.

You can switch to output mesh manager using tab control at the bottom of the manager or the renderer (Figure A.1). You will see identical manager panel with the same options but different meshes (Figure A.2).

You can use all functions from input manager, such as loading older outputs from hard drive for their comparison with new ones, but the main functionality here is the *Save Mesh...* button that allows you to store outputs of this application.

A.3.2 Settings

Settings tab of the left bottom panel of the main window (Figure A.1) is the initially selected one. It contains three main groups of settings.

Cache

The *Cache* settings allow you to turn on and off caching of individual execution steps. This speeds up repeated filtering of same inputs and allows you to faster switch between target steps and see how the method works (see sec A.3.4). The *All caches* option then activates or deactivates all caching options at once.

You can also delete all those caches from your hard drive using the *Clear caches* function.

There is also one additional caching that is used always and it stores coarse meshes. You can only clear this cache using *Clear coarse meshes* button.

All caches are stored in the *cache* folder in installation directory.

Parametrisation

In the *Parametrisation* group, spherical parametrisation variant of method can be activated by unchecking the *Direct morphing* option.

You can then specify if you want to use cascade modification of spherical parametrisation using the *Cascade parametrisation* checkbox.

You can also choose a relaxation method for the initial projection. There are two methods available:

- *Alexa 2000* [1]
- *Zhu 2009* [27]

Morphing

Morphing group allows you to modify final morphing interpolation step.

Input meshes are morphed by default. You can choose to morph output of non-rigid *ICP* alignment instead by checking the *Morph unified meshes* option.

You can also disable usage of reliability weights by unchecking the *Use reliability weights* options. Average coefficients are then used even for damaged meshes.

See Chapter 5.6.1 for more details.

A.3.3 Renderer interaction

There are two different renderers in the main application window (see Figure A.1 and Figure A.2).

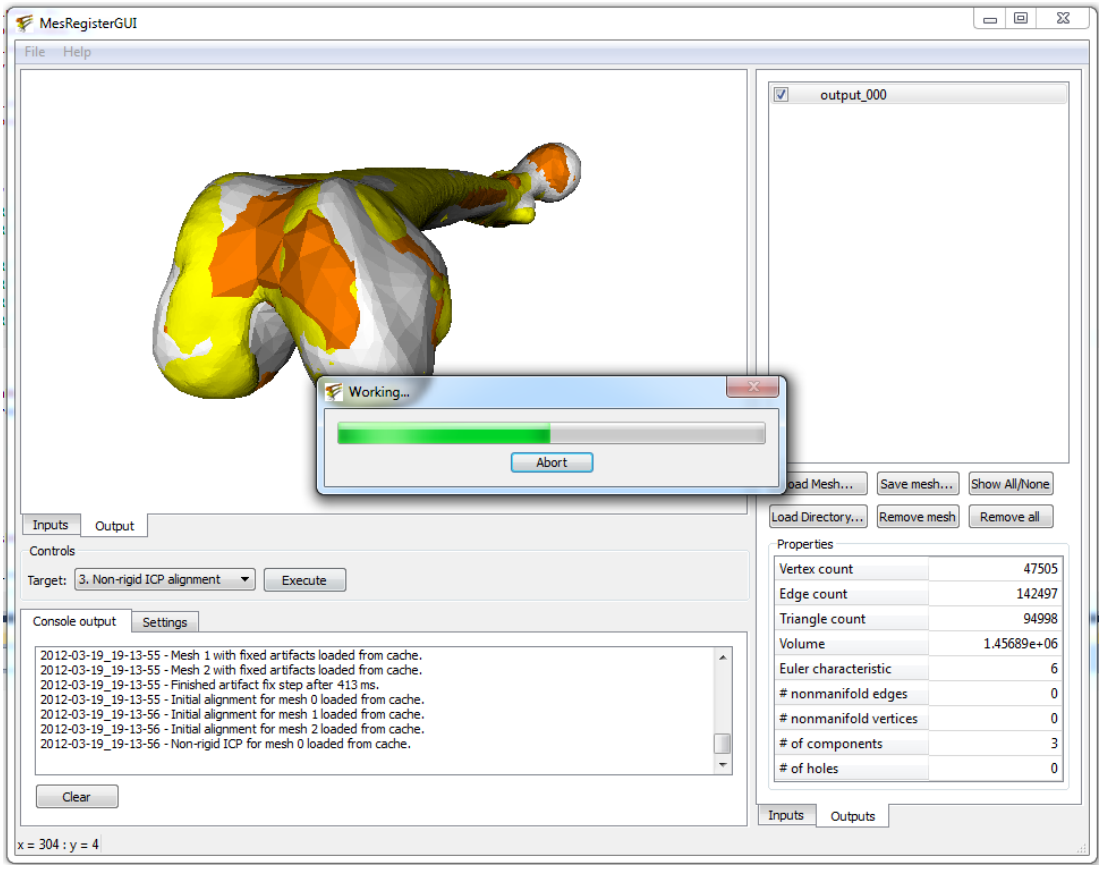


Figure A.2: Main window of the MeshRegisterGUI application with output of partial execution and progress bar.

The input renderer has black background for easier distinguish. The meshes here

are white solid by default or red wireframe if selected. You can either select mesh using mesh manager or you can just use your left mouse button and click on it in the renderer. This also causes the output console in the bottom part of the main window (Figure A.2) to give information about the mesh, clicked vertex and adjacent triangles.

The output window has white background and the characteristic of the mesh depends on the target state chosen (see sec. A.3.4).

You can always choose between the solid and wireframe rendering using the *S* and *W* buttons. The other standard *VTK* controls work here as well. An example is the *R* to reset camera. Others can be found in [10].

Standard camera control is mapped to mouse and its buttons. You can use *left mouse button drag* to rotate camera around the rotation centre. It is positioned in the centre of scene by default.

You can use *middle mouse button drag* to move the centre of rotation. And finally *right mouse button drag* allows you to zoom in and out.

Further instructions and alternative controls can again be found in [10].

A.3.4 Execution

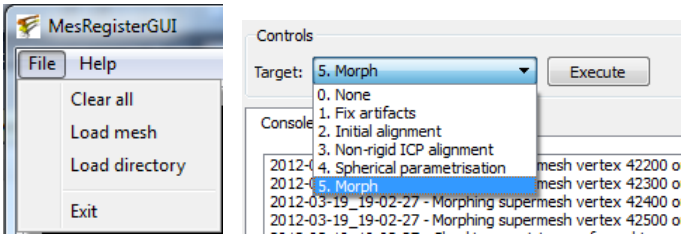
Before you can execute the morphing, be sure to have at least one input mesh selected and displayed in the input renderer (Figure A.1). Only then will the button *Execute* in the left middle part of the main window be enabled. You will usually want to specify more than one input. If only one input exists, all steps except the artifact removal become irrelevant.

Before you run the morphing, you can select target state in the select control just on the left. You can see six options here (see Figure A.3b):

- *0. None* - only merges inputs without any modification
- *1. Fix artifacts* - removes non-manifold edges and vertices, isolated components and fills holes. Produces wireframes distinguished by contrast colours.
- *2. Initial alignment* - does only rough alignment using rigid *PCA* based method. Produces aligned and coloured solids.
- *3. Non-rigid ICP alignment* - does precise alignment using non-rigid *ICP* based method. Produces aligned and coloured solids.
- *4. Spherical parametrisation* - creates spherical parametrisation of meshes. Produces wireframes of parametric domains distinguished by contrast colours.

- *5. Morph* - morphs the meshes together. Produces single solid mesh.

All steps of course perform execution of the previous steps.



(a) Main menu (b) Target selection

Figure A.3: Main menu and target of execution selection of the MeshRegisterGUI application.

When you finally click the *Execute* button, progress dialog with label *Working...* will appear. You can see progress on the progress bar and you can also abort the execution using *Abort* button. The GUI remains interactive during the execution.

The start of morphing also automatically switches to console view, so you can see the outputs of the method and individual steps. The console also contains both overall and partial timing information.

After the process is over, the pop-up window automatically closes and the perspective is switched to output view, so you can see the result straight away.

A.3.5 Others

Main menu

The top of the main window contains the main menu. It has only two main groups.

The *File* group contains three options (Figure A.3a):

- *Clear all* - clears all meshes in both input and output mesh managers
- *Load mesh* - opens load mesh from file dialog for the input mesh manager
- *Load directory* - opens load all meshes from directory dialog for the input mesh manager
- *Exit* - terminates the application

The item *About* in the *Help* menu opens an *About dialog* with information about application and its author.

Logging

All messages seen in the output console text area (Figure A.2) are simultaneously written to log files on hard drive. They can be found in the *logs* folder of the application installation directory.

They are named in form

`log_[date]_[time].txt`

for regular log messages or

`error_[date]_[time].txt`

for error messages.

This can be useful in case of fatal failure of the application.

Standard system console can additionally be also enabled using the symbol `USE_CONSOLE` in the *main.cpp* module of *MeshRegisterGUI* project. That however requires compilation of the application as is only intended for developers and debugging purposes.

Appendix B

Programmer documentation

B.1 SW components

B.1.1 GUI framework Qt

Qt is cross-platform application and GUI C++ framework from *Nokia* that helps to create applications faster and possible to move across different platforms provided that other parts of application are platform independent as well [7]. For logo see B.1. Its support is not only limited to desktop computers but also portable devices like smartphones.

Figure B.1: Logo of *QT* framework. Taken from [7].

I used it to build simple GUI for my application. In my bachelor work, I have worked with application based on *MAF* framework and more specifically its extension *Medical* for medical applications [13]. However I received a complete application at the beginning and only modified inner logic. The *MAF* framework is more complicated to work with and *Medical* is not available to public usage. I also had previous experiences with *Qt* and I have found out that is supported by

later described *VTK* library. It also comes with graphical editor and integration to *MS Visual Studio* so it saved me time necessary for GUI construction and I could focus on the main problem. The simplicity is also the reason why I did not use *Windows API* to create GUI by myself.

Qt is built from individual graphical components such as buttons, forms, text fields etc. just like any other similar framework. Hierarchical structure is built by inserting one to each other. The layout is given by special components called *layouts*. They support standard behaviour like float form side to side, extension to full space or grid distribution.

Actions are implemented using *slots*, *signals* and their mutual connections. For example button from class *QPushButton* provides signal *clicked* that can be connected to slot of application *QApplication* called *quit*. This means that if the button is clicked, *quit()* method of *QApplication* is called and application is terminated.

Here is an example of such code in C++ showing also usage of vertical layout (taken from [7]):

```
#include <QtGui>

int main(int argv, char **args)
{
    QApplication app(argv, args);

    QTextEdit *textEdit = new QTextEdit;
    QPushButton *quitButton = new QPushButton("&Quit");

    QObject::connect(quitButton, SIGNAL(clicked()),
                qApp, SLOT(quit()));

    QVBoxLayout *layout = new QVBoxLayout;
    layout->addWidget(textEdit);
    layout->addWidget(quitButton);

    QWidget window;
    window.setLayout(layout);

    window.show();

    return app.exec();
 }
```

Qt is not limited to *GUI* only. It also provides support for file system operations such as directory listing and threads synchronisation mechanisms like mutexes, semaphores or thread management. Although these features come useful, I did not use them in my filter itself and I limited their usage to GUI part only. The reason is to keep the kernel independent on *Qt* on source code level. It is relatively

large library with size of several hundred megabytes and takes up to few hours to compile. You can download binary distribution but there was limited support for *Visual Studio 2010* when I started with implementation (2011).

B.1.2 Visualisation system VTK

The Visualization Toolkit (*VTK*) (logo B.2) is multi-platform framework for data visualisation developed as opensource since 1993 [23].

Figure B.2: Logo of *VTK*. Taken from [10].

It is written in *C++*, but it supports many other languages such as *Java*, *Tcl*, *Perl* and *Python* through wrappers. I will use it in my *C++* application. It is distributed in form of source code which can be easily build on target platform using *CMake* ([16]). This provides both wide portability but also possibilities for different configuration. In my case, I enable support for *Qt*. Special classes like *QVTKWidget* are then generated that can be placed to *Qt* frames and hold *VTK* target render area. Therefore complicated GUI can be built by means of *Qt* leaving only the 3D rendering to *VTK*.

I have chosen this framework because I have experiences with it from my bachelor thesis [13] and I know that it provides lot of functionality I would have to implement myself otherwise. It is also compatible with the main part of SW developed in *VPHOP* project [26].

Architecture

VTK's main purpose is taking existing data and displaying them in graphical window for user interaction, which is mainly navigation in 3D-space including zooming. However to support this basic functionality, *VTK* contains many other classes. They handle both input and output operations with *VTK*'s own data format. They also provide representation of data classes with basic interaction and accessors.

For instance, there is a *vtkPolyData* which represents surface boundary object,

in my case triangular mesh. Its hierarchical structure can be seen in Figure B.3. Main structure hidden inside is simple vector of individual points *vtkPoints*. Topology of triangles is stored in cell lists of each point stored in external structures. Therefore you have to call special methods such as *BuildCells()* and *BuildLinks()* that ensures that secondary structures for mesh navigation are built. This then enables more complicated queries such as cell neighbours search.

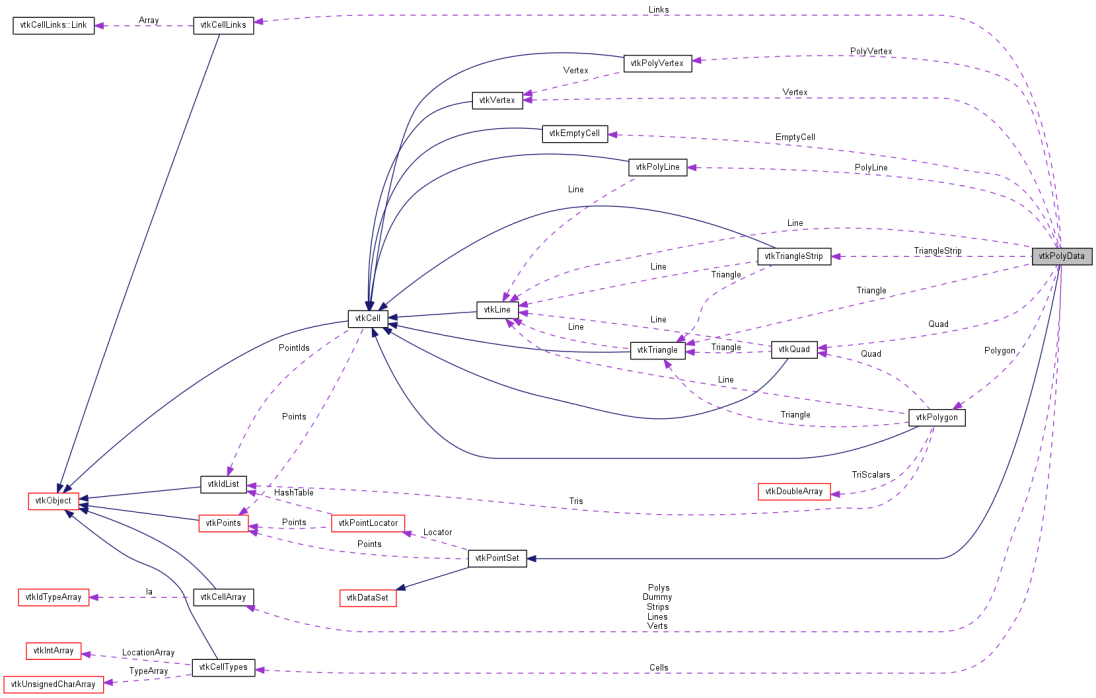


Figure B.3: Collaboration diagram for class *vtkPolyData*. Taken from [10].

The way of representation also means that methods like *GetCell()*, that gets object of single mesh triangle, do not reference permanent mesh data. They always built structures on demand. There are usually two versions of such methods. One writes to internal buffer, second to user provided memory. It is important to notice the difference if you program parallel application. The first version uses shared memory for all calls. Therefore if more threads accesses the method in same time, memory is overwritten and the method is not thread safe. This is also mentioned in documentation [10].

In addition to mostly passive data classes, there are also many filters. The idea of filters is to build pipe-line by connecting one to each other. Pipeline usually begins with data provider, which can be file reader or geometry generator. Filter outputs are then connected with next filter inputs so that they process what previous filter produced. The pipeline usually ends in *vtkRenderer* which displays data wrapped in *actors* and *mappers*. These are the ones which actually know

how to visualise different structures from meshes or point clouds to volumetric data.

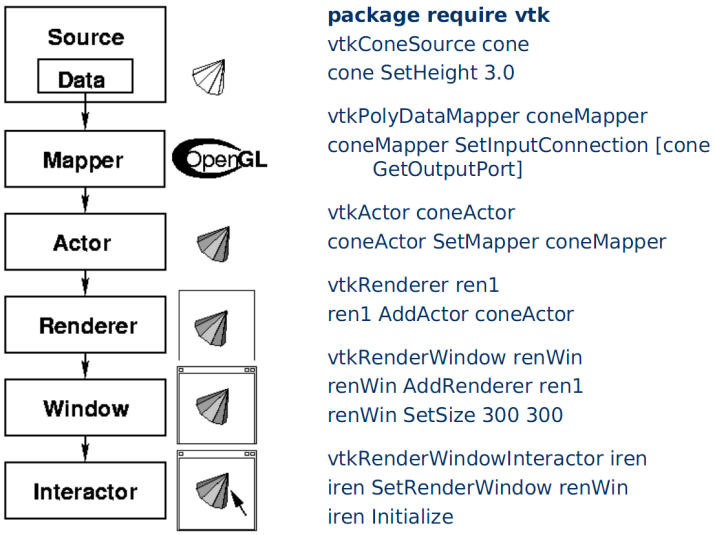


Figure B.4: Example of simple graphical pipeline with *VTK*. Generates and displays cone. Implemented in scripting language *Tcl*. Taken from [9].

Notable filters

vtkFillHolesFilter This filter processes input meshes and finds holes in their surface by detecting their boundaries. Then it simply triangulates those creating new edges in the topology. It is possible to set how large the filled hole can be so that it does not triangulate space between distant object if this could occur.

vtkFeatureEdges This class traverses through mesh edges and filters only those with specified features. Usage is limited to detection of edges with specified number of adjacent triangles and angles. It can be useful for detection of non-manifold edges as well as holes.

It gets useful as *vtkPolyData* itself does not provide natural way of edge access like for points and triangles. There is no method that gets number of edges as well as method that gets single edge by ID. You can only access edges based on end points or you could also use cell traversal and process boundaries of each triangle. That would however inevitably lead to redundant check of shared edges. This filter makes work with edge structures much more convenient.

vtkQuadricDecimation Along with similar filters *vtkDecimate*, *vtkDecimatePro* and *vtkQuadricClustering* this filter provides technique for reduction of mesh size in a matter of vertex count. It was originally used in my bachelor work [13] to create coarse outer hull for mean value coordinate calculations *MVC*. However it proved to have problems with thin features and most importantly it does not guarantee the product mesh to be outer envelope. This reduced precision of *MVC* calculation [13]. For this reason, it was replaced by newly created filter described in Section B.1.3.

B.1.3 Progressive hull filter

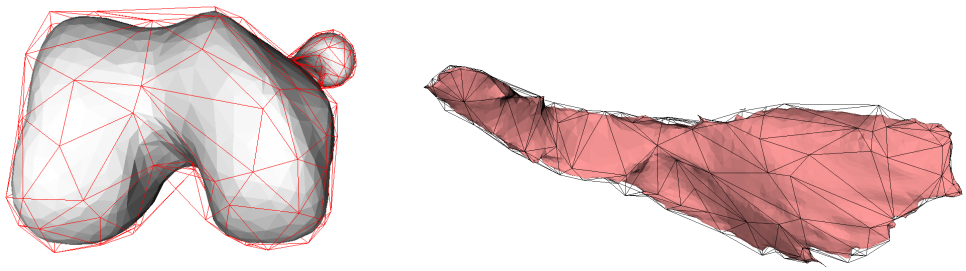
An author's implementation of progressive hull construction method proposed by *Bc. David Cholt* and described in the article [6] was used to create coarse meshes in several parts of my algorithm implementation. The same code is also used in the improved version of the deformation filter [17].

The main class *vtkProgressiveHull* of the *C++* code is inherited from *vtkPolyDataToPolyDataFilter*, so it is used as standard *VTK* filter.

The method itself is based on an edge decimation approach. Edges are sorted according to the volume gain caused by their removal and consequently each of them replaced by single new vertex. This vertex is in such position that the new mesh is an envelope of the previous and it also has minimum possible volume.

The article [6] describes changes made to an older method that ensure more equal distribution of decimation over the surface, better stability in spiky regions of the mesh and no creases in the output mesh. This is very useful for jagged meshes like one of the Sartorius muscle models (see Figure D.3).

An example of the output can be seen in Figure B.5.



(a) Femur bone (b) Iliacus muscle

Figure B.5: Examples of coarse meshes with target size of 300 vertices created using filter described in sec. B.1.3. Coarse mesh displayed as outer progressive hull of the input high-polygonal mesh.

B.1.4 3D Embedding by Parus

3D Embedding is experimental implementation of Alexa's spherical parametrisation [1] created by *Ing. Jindřich Parus, Ph.D.* I have used it as reference for my own implementation and it helped me to check correctness. It was originally realised as a *Delphi* GUI application. I created *DLL* to be able to use it directly from my C++ application. However I later chose to rewrite the implementation completely according to original article [1] in order to be able to fully control the process. I therefore used this implementation mainly for validation purposes.

B.2 Implementation details

Most important code of algorithm implementation can be found in *ExtendedMesh* and *vtkMeshRegister*.

ExtendedMesh is main mesh data structure. It uses embedded *vtkPolyData* for topology information but contains copy of vertices for faster access. It also adds many new methods for accessing inner centre, point neighbours, main axes, spherical projection and so on. Additionally it supports import and export to HDD as well as visualisation using various colours.

vtkMeshRegister is the main class of the implementation. It is child of *vtkPolyDataAlgorithm* and it executes the mesh registration and morphing algorithm and implements parts that work on more than one mesh.

It supports clever caching of partial results enabling much faster development of individual steps by skipping the previous ones and loading their results from hard drive. This is vital as performance in interactive debug mode gets often very low. Caching is enabled through class *MeshCache* and it uses simple approach to generate string filename from *vtkPolyData* key to store *vtkPolyData* or general memory data block value.

vtkMeshRegister also supports premature termination after any major step to show partial results and provide tool to tune them separately. This feature is accessible from GUI.

To keep caller interactive, asynchronous abort calls are supported and progress information is send to listeners during execution. GUI uses this to show progress bar and provide *Abort* button (see Figure A.2).

There are also some other interesting aspects in the implementation I would like to emphasise.

Logger class for HDD and console logging uses smart C++ macros to accept *std::stringstream* inputs in extremely simple way. Only this one row has to be

added to log almost any information:

```
LoggerSs("The result is " << result << ".");
```

This makes logging very convenient and easy to use. Additionally, it also logs into console through *std::cout*. The GUI part then redirects this stream into *QPlainTextEdit* GUI text editor. This allows communication with user even if the console is not shown. However application supports native console activation even in GUI mode.

I have also adopted auto-pointer principle that is represented by *vtkSmartPointer* template in *VTK*. I have made my own implementation to keep it usable without *VTK* in different projects.

ReferencedPointer is my alternative to *vtkSmartPointer*. It requests template argument to implement interface *IReferenced*. Then objects are created in similar way as in *VTK* would. Only difference is that I support parameters in factory methods and therefore composed calls are used to create auto-pointer. Therefore where the code

```
ExtendedMesh* mesh = ExtendedMesh::New(polyData);
mesh->Delete();
```

could be used, is replaced by safer

```
ReferencedPointer<ExtendedMesh> mesh =
          ReferencedPointer<ExtendedMesh>::New(
                  ExtendedMesh::New(polyData));
```

alternative. On the top of convenience of automatic destructor call, it prevents memory leaks as the destructor is called even after exception, because it is tied with stack shrink.

In *VTK*, the code would look like this:

```
vtkSmartPointer<ExtendedMesh> mesh =
               vtkSmartPointer<ExtendedMesh>::New();
mesh->SetPolyData(polyData);
```

I use many custom structure for basic geometry elements such as *Vertex3*, *Vertex4*, *Matrix4x4*. They are far more convenient to use than *VTK*'s raw double arrays as they incorporate algebraic operations through both static and instance methods and overloaded operators. They also enable stream text output.

These structures were used in some of my previous projects as well, but were improved in this version. They are now fully templated and therefore support both `float` and `double` as well as `integer` types. Own type was also defined as `pkFloat` derived from `double` to make change of main project data type even easier. Double was chosen for better precision plus because it is default data type in *VTK*.

Usage of templates is made simpler by defined data type synonyms such as *Vertex4d* for *double* parametrised *Vertex4* or *Vertex4t* for *pkFloat* and so on.

B.3 Architecture

In the *Microsoft Visual Studio 2010* solution *MeshRegister.sln* there are two main projects.

I will provide brief description of their requirements and contained classes. More details can be found in source code documentation. The general discussion of implementation was provided in Section 5.1.

B.3.1 MeshRegister project

Project *MeshRegister* contains implementation of the algorithm and requires *VTK 5.8.0* or newer to build.

The context of classes and their relations can also be seen on UML class diagram in Figure B.6.

Libraries

- *VTK* - provides data structures, algorithms and minor I/O support

Interfaces

- *IIdentity* - interface of object identifiable by numeric ID
- *IReferenced* - interface of object with reference counter

Classes

- *BigMatrix* - sizable matrix of simple data types
- *BigMatrixReferenced* - sizable matrix of objects with referenced counter. Supports auto-pointer mechanism.
- *CoarseMesh* - generator of coarse mesh. Also calculates mutual *MVC* and supports back projection from coarse to full size mesh.
- *Embedding3D* - singleton wrapper for Parus' *3D embedding* DLL B.1.4

- *ExtendedMesh* - main data structure for mesh. Wraps *vtkPolyData*, adds new methods, manipulations, faster access to data, normal caching and many more.
- *IdentityBase* - default realisation of *IIdentity*
- *Logger* - smart HDD and console logger. Accepts streams to easily combine numeric values with text.
- *MathMatrix* - child of *BigMatrix* supporting mathematical matrix operations such as addition, multiplication, transposition and inversion.
- *MyMath* - static mathematic library for intersection, barycentric coordinates etc.
- *MyVector* - dynamic vector structure with multi-threading support
- *OOBox* - object oriented bounding box for *ExtendedMesh*
- *PriorityQueue* - support class for *vtkProgressiveHull*
- *ReferencedBase* - default realisation of *IReferenced*
- *StdRedirector* - redirector of *std::cout* and *std::cerr*
- *Utils* - static utility class
- *VertexNeighbours* - cache for faster access to vertex neighbours
- *vtkDamage* - child of *vtkPolyDataAlgorithm*, filter that generates non manifold mesh from source mesh for testing purposes. Adds non manifold edges, vertices, creates hole and inserts isolated components.
- *vtkMeshRegister* - child of *vtkPolyDataAlgorithm*, implementation of main algorithm and its major steps
- *vtkProgressiveHull* - filter for construction of coarse mesh hull using edge decimation. Child of *vtkPolyDataToPolyDataFilter*. Made by *Bc. David Cholt*. See Section B.1.3.
- *vtkProgressiveHullCPU* - CPU based realisation of *vtkProgressiveHull*. Made by David Cholt. See Section B.1.3.
- *vtkProgressiveHullCUDA* - GPU based (*CUDA*) realisation of *vtkProgressiveHull*. Made by *Bc. David Cholt*. See Section B.1.3.

Structures

- *Color4* - normalised RGBA color structure
- *Matrix4x4* - template of 4×4 matrix with algebraic operations and transformation factories
- *MemoryDisposer* - structure for automatic disposing of memory blocks after stack shrink
- *MutexLocker* - structure for automatic unlocking of mutexes after stack shrink. Uses *vtkMutexLock*.
- *MyTimer* - timer with nano-second precision. Uses *Windows API*.
- *Plane* - template of plane in 3D space, holds general equation of plane as *Vertex4*
- *ReferencedPointer* - auto-pointer for *IReferenced* objects
- *ValueCache* - memory cache of simple value with invalidation support
- *Vertex3* - template of three dimensional vector with algebraic operations
- *Vertex4* - template of four dimensional vector with algebraic operations

Other

- *Common.h* - header with basic definitions and common includes
- *my_types.h* - custom data types

B.3.2 MeshRegisterGUI project

Project *MeshRegisterGUI* contains graphical user interface for the *MeshRegister* and requires the *MeshRegister* project, *VTK 5.8.0* or newer and *Qt 4.7.4* to build.

The context of classes and their relations can also be seen on UML class diagram in Figure B.7.

Libraries

- *MeshRegister* - provides implementation of muscle registration and morphing
- *VTK* - provides visualisation of 3D data and access to data structures used in *MeshRegister*
- *Qt* - provides GUI, I/O operations and thread management

Interfaces

- *IResultAcceptor* - interface accepting a result of *vtkMeshRegister* execution

Classes

- *AboutDialog* - dialog with information about the application and author, child of *QDialog*
- *FilterWorker* - parallel thread for asynchronous execution of *vtkMeshRegister*, child of *QThread*
- *MeshData* - data class holding *ExtendedMesh* instance together with statistics and visual properties
- *MeshManager* - widget for mesh management, child of *QMainWindow*
- *MeshRenderer* - widget for mesh rendering, child of *QVTKWidget*
- *MeshRegisterGUI* - main window of application, child of *QMainWindow*
- *ProgressDialog* - dialog with filter progress information and abortion option, child of *QDialog*
- *QStdRedirector* - class for redirection of standard output streams to *QPlainTextEdit* using *StdRedirector*.

Other

- *main.cpp* - module with application's entry point. Opens the main window *MeshRegisterGUI*.

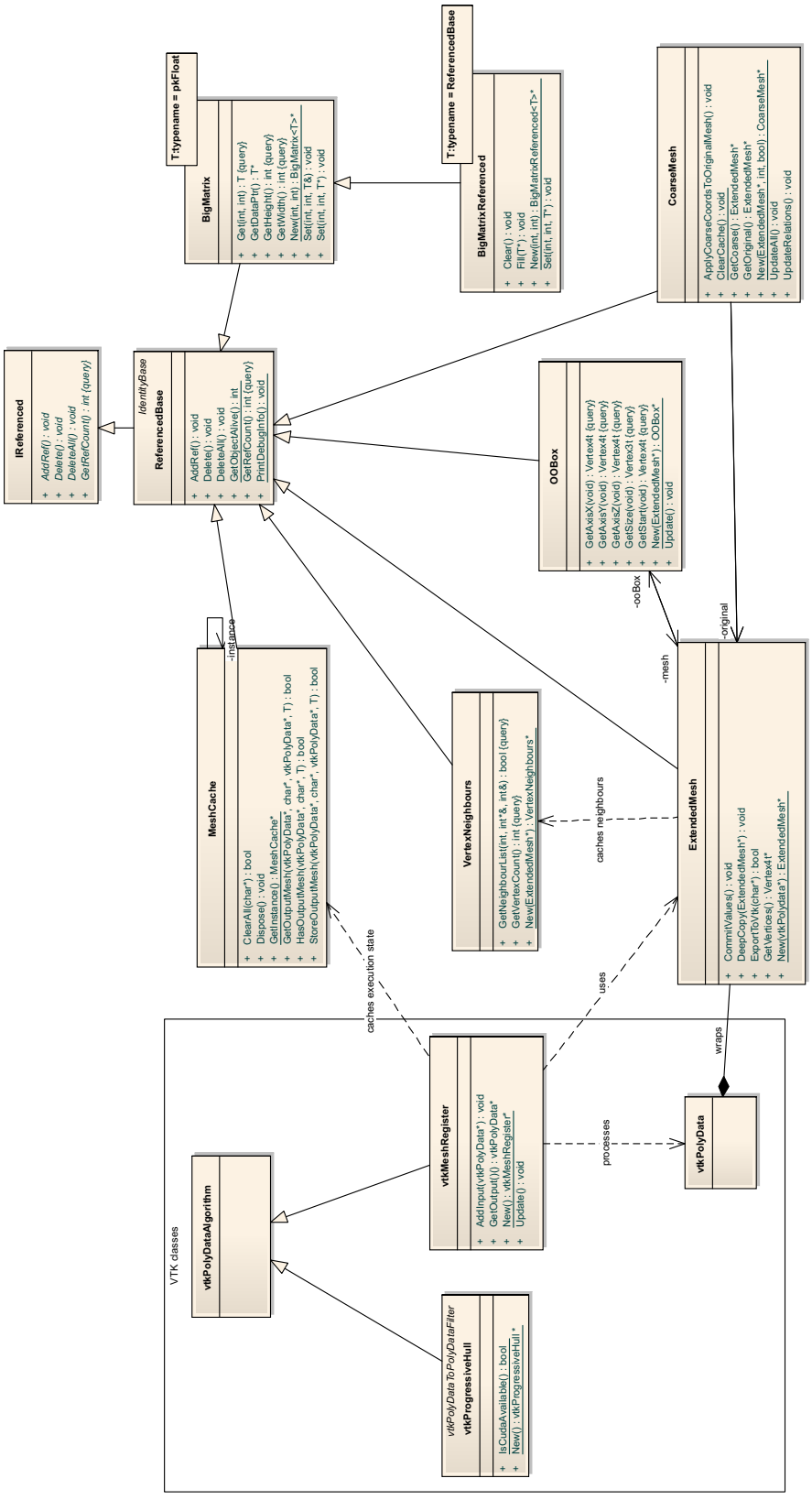


Figure B.6: UML class diagram of MSVS project MeshRegister.

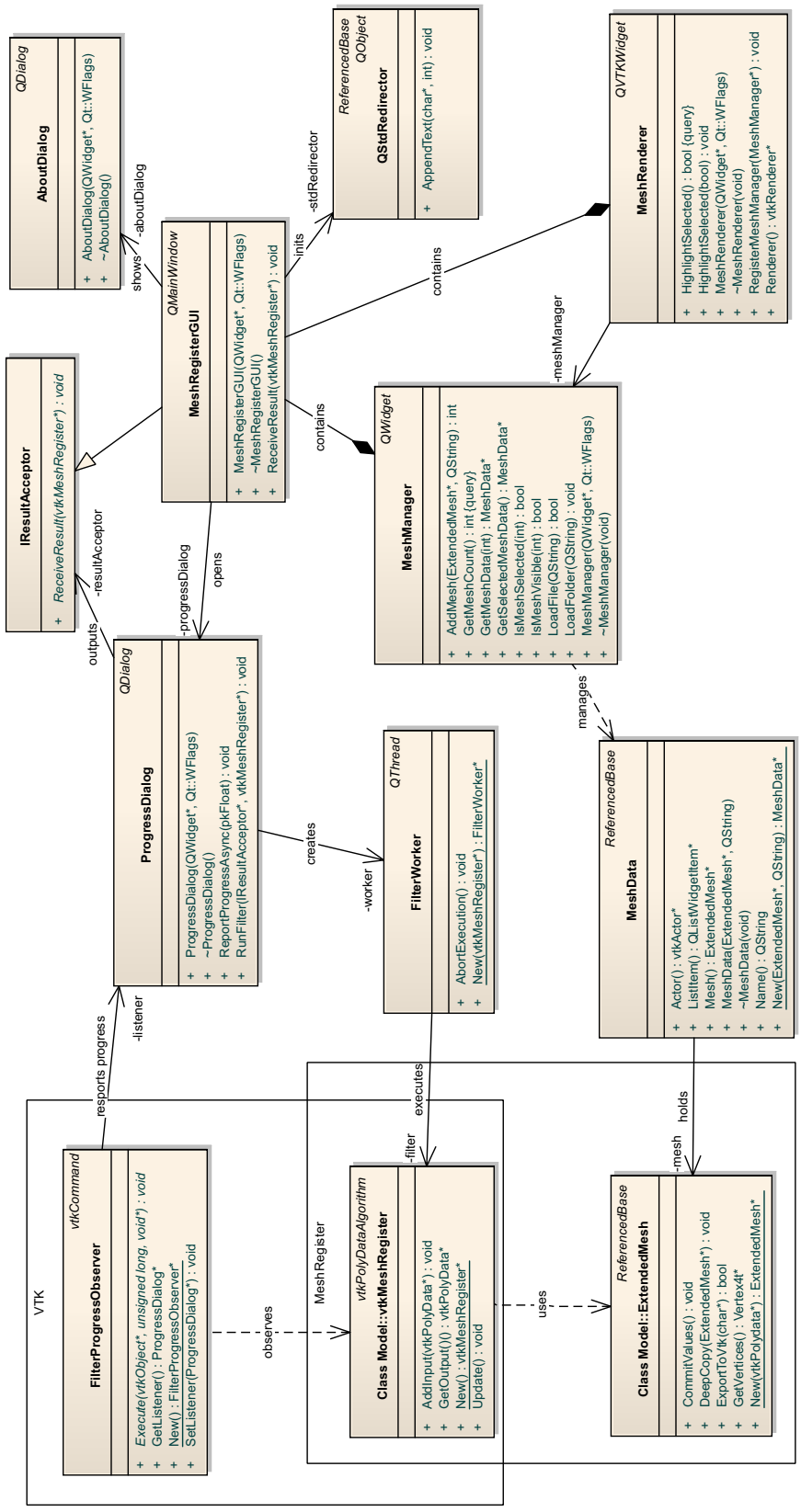


Figure B.7: UML class diagram of MSVS project MeshRegisterGUI.

Appendix C

Algorithms

This chapter provides partial algorithm description referenced from the Chapter 4.

The common variables used are similar to those used in the main text where detailed explanations, discussions and references can be found.

Their brief summary follows:

i ... general index

k ... index of mesh — forall $k \Rightarrow$ for each mesh

X_k ... k-th input mesh

$|X_k|$ ... size of k-th mesh in number of vertices

$\overrightarrow{x_i}$ ... i-th vertex

P_i ... parametrisation of i-th mesh

$\overrightarrow{x_i}^P$ ... parametrisation of i-th vertex

T_i ... i-th triangle

e_i ... i-th edge

S ... supermesh

r_i ... reliability of i-th triangle

f_i ... triangle fan of i-th vertex

C_i ... i-th component of mesh

H_i ... i-th hole in mesh

FP_i ... parametrisation of i-th mesh

$\overrightarrow{\Lambda}^S_{i,k}$ ... spherical barycentric coordinates of i-th vertex in k-th parametric domain

C.1 Main steps

Algorithms C.1, C.2, C.4 and C.5 describe individual main steps of the global algorithms 4.1 and 4.2.

C.2 Auxiliary algorithms

Auxiliary algorithms C.7, C.8, C.9, C.12, C.13 and C.14 describe some elementary yet non-trivial steps used in above referenced main step algorithms.

All algorithms are cross referenced with their text description in Chapter 4.

Algorithm C.1 Fix of artifacts in meshes (see Chapter 4.2)

for all input mesh X_k **do**
  for all edge e_i in X_k **do**
    if e_i has more than 2 triangles **then**
      Remove all triangles of e_i {Removes non-manifold edges}
    end if
  end for
  for all vertex $\overrightarrow{v_i}$ in X_k **do**
    Find triangle fans for $\overrightarrow{v_i}$
    Pick largest closed fan f_c {if exists}
    for all fan f_i **do**
      if $f_i \neq f_c$ **then**
        Remove all triangles in f_i {Removes vertices with multiple fans}
      end if
    end for
  end for
  Find all components $C_{k,i}$ of X_k using *DFS*
  $C_{k,max} \leftarrow$ largest $C_{k,i}$ in vertex count
  for all $C_{k,i}$ **do**
    if $C_{k,i} \neq C_{k,max}$ **then**
      Remove all triangles in $C_{k,i}$ {Removes minor components}
    end if
  end for
  Find holes $H_{k,i}$ in X_k
  $\tilde{T}_k \leftarrow \emptyset$
  for all $H_{k,i}$ **do**
    Triangulate hole $H_{k,i}$
    $\tilde{T}_k \leftarrow \tilde{T}_k \cup$ triangles to fill $H_{k,i}$
  end for{State reliabilities $r_{k,i}$}
  for all triangle t_i in X_k **do**
    if $t_i \in \tilde{T}_k$ **then**
      $r_{k,i} \leftarrow 0$
    else
      $r_{k,i} \leftarrow 1$
    end if
  end for
end for
return $\{X_k\}$

Algorithm C.2 Mesh alignment using PCA (see Chapter 4.3.1)

Pick target mesh X_T
Find main axes and centre of X_T (more in alg. C.7)
for all input mesh X_k **do**
  Find main axes and centre of X_k
  Find translation and rotation matrix to align X_k to X_T (more in alg. C.8)

  Apply transformation to X_k
  for all axis orientation **do**
    Rotate X_k by axis
    Measure distance between X_k and X_T vertices (more in alg. C.9)
    Remember nearest axis
  end for
  Transform X_k by minimal axis
end for
return $\{X_k\}$

Algorithm C.3 Non-rigid mesh alignment using modified ICP based method from [5] (see Chapter 4.4).

Select target mesh Y
for all input mesh X_k where $X_k \neq Y$ **do**
  $r \leftarrow$ number of feature points
  for $j = 0 \rightarrow r - 1$ **do**
    $FP_j \leftarrow$ randomly picked point on mesh X_k
    $S_j \leftarrow$ ICP region for FP_j {See alg. C.10 for details.}
    Run ICP to align S_i to Y {See alg. C.11 for details.}
    $M_j \leftarrow$ total transformation matrix from ICP
  end for
  for all vertices $\overrightarrow{x_i}$ in X_k **do**
    $W \leftarrow 0$
    for all $j = 0 \rightarrow r - 1$ **do**
      $d \leftarrow$ distance between $\overrightarrow{x_i}$ and $\overrightarrow{fp_j}$ {Euclid or geodesic distance can be used, see 4.4.3 for details.}
      $d \leftarrow d/(MESH\_SIZE \cdot d_{max})$ {Normalise maximum allowed range to 1}
      $w_j \leftarrow 1 - d^{1.5}$
      $W \leftarrow W + w_j$
    end for
    $M \leftarrow \boldsymbol{I}$ {4×4 identity matrix}
    for all $j = 0 \rightarrow r - 1$ **do**
      if $W > 0$ **then**
        $w_j \leftarrow w_j/W$ {Normalise weight sum}
      else
        $w_j \leftarrow 1/r$ {Alternative uniform distribution to remote vertex}
        $M \leftarrow M + w_j \cdot M_j$ {Interpolate transformation matrices}
      end if
    end for
    $\overrightarrow{x_i} \leftarrow M \cdot \overrightarrow{x_i}$ {Deform vertex $\overrightarrow{x_i}$}
  end for
end for
return $\{X_k\}$

Algorithm C.4 Cascade spherical parametrisation (see Chapter 4.5.3)

$mesh_{max} \leftarrow$ maximum number of vertices in input mesh
$size_0 \leftarrow size_{last} \cdot c^{\lfloor \log_c \frac{mesh_{max}}{size_{last}} \rfloor}$ {See sec. 4.5.3}
$size \leftarrow size_0$
for all input mesh X_k **do**
  $X_{k,0} \leftarrow X_k$ {Start with input meshes}
end for
$i \leftarrow 0$ {Decimate down}
while $size \geq size_{last}$ **do**
  for all k **do**
    $X_{k,i+1} \leftarrow$ decimate $X_{k,i}$ to $size$
    Calculate MVC for $X_{k,i+1}$ in $X_{k,i}$ (more in article [24])
  end for
  $size \leftarrow size/c$ {Scale down}
  $i \leftarrow i + 1$
end while
for all k **do**
  $P_{k,i} \leftarrow X_{k,i}$ {Initialise parametrisation with smallest mesh}
  Find inner centre of $P_{k,i}$ (more in alg. C.12)
  Move $P_{k,i}$ by inner centre to $(0, 0, 0)$
  Project $P_{k,i}$ to sphere
  Relax $P_{k,i}$ (more in alg. C.13)
end for
{Interpolate up}
while $size \leq size_{last}$ **do**
  $size \leftarrow size \cdot c$ {Scale up}
  for all k **do**
    $P_{k,i-1} \leftarrow$ interpolation of $P_{k,i}$ using MVC (more in [13])
    Project $P_{k,i-1}$ to sphere
    Find feature points FP_k
  end for
  Average feature points $FP_{avg} \leftarrow avg(P_{k,i-1})$ for all k
  for all k **do**
    shift of $FP_k \leftarrow FP_k - FP_{avg}$
    Distribute shift of FP_k to surface of $P_{k,i-1}$ C.14
    Relax $P_{k,i-1}$
  end for
  $i \leftarrow i - 1$
  Output $P_{k,i}$ as parametrisation of X_k
end while
return $\{P_{0,0}, P_{1,0}, \dots\}$

Algorithm C.5 Multi-morphing of meshes X_k to supermesh S using spherical parametric domains (see Chapter 4.6)

$k_S \leftarrow \arg\max_k \sum_i r_{k,i}$ {Choose supermesh based largest on reliability sum}
$S \leftarrow X_{k_S}$ {Copy supermesh structure for output}
for all $i = 0 \rightarrow |S|$ **do**
  $l_i \leftarrow 0$
  for all k **do**
    $T_{k,j} \leftarrow$ spherical triangle of X_k^P where the $\overrightarrow{s_i}^P$ lies
    $\overrightarrow{\Lambda}^S_{i,k} \leftarrow$ spherical barycentric coords of $\overrightarrow{s_i}^P$ in $T_{k,j}$
    $l_{i,k} \leftarrow r_{T_{k,j},i}$ {Reliability of target triangle}
    $l_i \leftarrow W + l_{i,k}$
  end for
  $\overrightarrow{s_i} \leftarrow (0,0,0)$
  for all k **do** {Calculate weights}
    $l_{i,k} \leftarrow l_{i,k}/l_i$ {Normalise linear coefficients}
    $(a,b,c) \leftarrow T_{k,j}$
    $\overrightarrow{s}_{i,k} \leftarrow \overrightarrow{\Lambda}^S_{i,k} \cdot (\overrightarrow{x}_{k,a}, \overrightarrow{x}_{k,b}, \overrightarrow{x}_{k,c})$
    $\overrightarrow{s_i} \leftarrow \overrightarrow{s_i} + l_{i,k} \cdot \overrightarrow{s}_{i,k}$
  end for
end for
return S

Algorithm C.6 Multi-morphing of meshes X_k to supermesh S directly using non-rigidly registered input meshes (see Chapter 4.7)

$k_S \leftarrow \arg\max_k \sum_i r_{k,i}$ {Choose supermesh based largest on reliability sum}
$S \leftarrow X_{k_S}$ {Copy supermesh structure for output}
for all $i = 0 \rightarrow |S|$ **do**
  $l_i \leftarrow 0$
  for all k **do**
    $T_{k,j} \leftarrow$ nearest triangle of $\tilde{X}_k$ from $\overrightarrow{s_i}$ (see 4.3.2 for point-triangle distance measure)
    $\overrightarrow{\Lambda}_{i,k} \leftarrow$ 3D barycentric coords of $\overrightarrow{s_i}$ in $T_{k,j}$
    if $\max \overrightarrow{\Lambda}_{i,k} = 1$ **then** {Nearest point lies on vertex of $\tilde{X}_k$}
      $T_{k,j} \leftarrow$ triangle of nearest point fan with highest reliability
      Update $\overrightarrow{\Lambda}_{i,k}$
    else if $\min \overrightarrow{\Lambda}_{i,k} = 0$ **then** {Nearest point lies on edge of $\tilde{X}_k$}
      $T_{k,j} \leftarrow$ triangle of nearest edge with highest reliability
      Update $\overrightarrow{\Lambda}_{i,k}$
    end if
    $l_{i,k} \leftarrow r_{T_{k,j},i}$ {Reliability of target triangle}
    $l_i \leftarrow W + l_{i,k}$
  end for
  $\overrightarrow{s_i} \leftarrow (0,0,0)$
  for all k **do** {Calculate weights}
    $l_{i,k} \leftarrow l_{i,k}/l_i$ {Normalise linear coefficients}
    $(a,b,c) \leftarrow T_{k,j}$
    $\overrightarrow{s}_{i,k} \leftarrow \overrightarrow{\Lambda}^S_{i,k} \cdot (\overrightarrow{x}_{k,a}, \overrightarrow{x}_{k,b}, \overrightarrow{x}_{k,c})$
    $\overrightarrow{s_i} \leftarrow \overrightarrow{s_i} + l_{i,k} \cdot \overrightarrow{s}_{i,k}$
  end for
end for
return S

Algorithm C.7 Finding main axes of mesh M (see Chapter 4.3.1)

$\overrightarrow{\mu} \leftarrow (0,0,0)$ {Centre of gravity point}
for all $\overrightarrow{x_i} in M$ **do**
  $\overrightarrow{\mu} \leftarrow \overrightarrow{\mu} + \overrightarrow{x_i}$
end for
$\overrightarrow{\mu} \leftarrow \overrightarrow{\mu}/|M|$
$A \leftarrow$ empty 3×3 matrix
for all $\overrightarrow{x_i} in M$ **do**
  $\Delta\overrightarrow{\mu} \leftarrow \overrightarrow{x_i} - \overrightarrow{\mu}$
  $A \leftarrow A + \Delta\overrightarrow{\mu} \cdot \Delta\overrightarrow{\mu}^T$
end for
$\Lambda \leftarrow$ eigenvalues of A
$V \leftarrow$ eigenvectors of A
$\overrightarrow{axis_x} \leftarrow V[\arg\max_i \Lambda]$
$\overrightarrow{axis_y} \leftarrow V[\arg|_i \Lambda]$
$\overrightarrow{axis_z} \leftarrow V[\arg\min_i \Lambda]$
return $\{\overrightarrow{axis_x}, \overrightarrow{axis_y}, \overrightarrow{axis_z}\}$

Algorithm C.8 Finding transformation of mesh M to basic pose (see Chapter 4.3.1)

$axis_x, axis_y, axis_z \leftarrow$ main axes of M (see alg. C.7)
$\overrightarrow{\mu} \leftarrow$ centre of M
$\overrightarrow{t} = -\overrightarrow{\mu}$
$T \leftarrow$ translation matrix for vector $\overrightarrow{t}$
$\alpha \leftarrow$ angle between $axis_x$ and $(1,0,0)$
$\overrightarrow{o_\alpha} \leftarrow$ rotation axis between $axis_x$ and $(1,0,0)$
$R_\alpha \leftarrow$ rotation matrix with angle α around axis $\overrightarrow{o_\alpha}$
$axis_y\ R_\alpha \cdot axis_y$
$\beta \leftarrow$ angle between $axis_y$ and $(0,1,0)$
$\overrightarrow{o_\beta} \leftarrow$ rotation axis between $axis_y$ and $(0,1,0)$
$R_\beta \leftarrow$ rotation matrix with angle β around axis $\overrightarrow{o_\beta}$
$A \leftarrow R_\beta \cdot R_\alpha \cdot T$ {Combine elementary transformations}
for all $\overrightarrow{x_i} in M$ **do**
  $x_i \leftarrow A \cdot x_i$ {Transform mesh by vertex}
end for
return M

Algorithm C.9 Distance metric for alignment quality test of meshes X_0 and X_1 (see Chapter 4.3.2)

$d \leftarrow 0$ {Total distance}
for $side = 0 \rightarrow 2$ **do** {Project mesh 0 to 1 and then vice-versa}
  $d_{side} \leftarrow 0$ {Distance from mesh 0 to mesh 1}
  for all $\overrightarrow{x}_{0,i}$ in X_0 **do**
    $j_{min} \leftarrow 0$
    $minSq \leftarrow \infty$
    for all triangle $T_{1,j}$ in X_1 **do**
      $\rho \leftarrow T^A_{1,j} \times T^B_{1,j} \times T^C_{1,j}$ {Plane of triangle $t_{1,j}$}
      $\overrightarrow{x}^{\rho}_{0,i} \leftarrow$ projection of $\overrightarrow{x}_{0,i}$ to ρ
      $d_{i,j}$
      if $\overrightarrow{x}^{\rho}_{0,i}$ lies in $T_{1,j}$ **then**
        $d_{i,j} \leftarrow |\overrightarrow{x}^{\rho}_{0,i} - \overrightarrow{x}_{0,i}|^2$ {Distance to perpendicular projection}
      else
        $d_{i,j} \leftarrow$ distance to nearest edge of $T_{1,j}$
      end if
      if $d_{i,j} < min$ **then**
        $minSq \leftarrow d_{i,j}$
        $j_{min} \leftarrow j$
      end if
    end for
    $d_{side} \leftarrow d_{side} + minSq$
  end for
  $d \leftarrow d + d_{side}/|X_0|$
  Swap X_0 and X_1
end for
$d \leftarrow d/2$
return d

Algorithm C.10 Selection of region $\boldsymbol{S}$ for feature point (FP) based on k-neighbourhood (see Chapter 4.4.1)

$\boldsymbol{S} \leftarrow \emptyset$
$\boldsymbol{S} \leftarrow \boldsymbol{S} \cup \{\overrightarrow{FP}\}$
for $i = 0 \rightarrow k - 1$ **do**
  $\boldsymbol{R} \leftarrow \boldsymbol{S}$
  while is not empty $\boldsymbol{R}$ **do**
    $\overrightarrow{x} \leftarrow$ pop element from $\boldsymbol{R}$
    for all neighbours $\overrightarrow{v_j}$ of $\overrightarrow{x}$ **do**
      if $\boldsymbol{S}$ does not contain $\overrightarrow{v_j}$ **then**
        $\boldsymbol{S} \leftarrow \boldsymbol{S} \cup \{\overrightarrow{v_j}\}$
      end if
    end for
  end while
end for
return $\boldsymbol{S}$

Algorithm C.11 Iterative schema of ICP aligning points from $\boldsymbol{S}$ to mesh Y (see Chapter 4.4.2)

$M \leftarrow \boldsymbol{I}$ {Initialise transformation matrix.}
$\overrightarrow{\mu_Y} \leftarrow$ centre of Y
loop
  $\boldsymbol{Z} \leftarrow \emptyset$
  for all $\overrightarrow{s_i}$ in $\boldsymbol{S}$ **do**
    $\boldsymbol{Z} \leftarrow \boldsymbol{Z} \cup \arg\min \overrightarrow{y_j} \in Y |\overrightarrow{s_i} - \overrightarrow{y_j}|^2$ {Get nearest points in target mesh.}
  end for
  $\overrightarrow{\mu_X} \leftarrow 1/|\boldsymbol{S}| \cdot \sum_i \overrightarrow{s_i}$ {Centre of $\boldsymbol{S}$}
  $\overrightarrow{\mu_Y} \leftarrow 1/|\boldsymbol{Z}| \cdot \sum_i \overrightarrow{z_i}$ {Centre of $\boldsymbol{Z}$}
  $\Sigma \leftarrow$ cross-covariance matrix (see equation 2.2)
  $Q \leftarrow$ matrix build from Σ according to equation 2.2
  Find eigenvalues and eigenvectors of Q
  $\overrightarrow{q} \leftarrow$ eigenvector for largest eigenvalue of Q
  $M_R \leftarrow \boldsymbol{R}(\overrightarrow{q})$ {See eq. 2.7.}
  $\overrightarrow{t} \leftarrow \overrightarrow{\mu_Y} - \overrightarrow{\mu_X}$
  $M_T \leftarrow$ translation matrix for vector $\overrightarrow{t}$
  $M \leftarrow M_T \cdot M_R \cdot M$
  $error \leftarrow 1/|\boldsymbol{S}| \cdot \sum_i |\overrightarrow{s_i} - \overrightarrow{z_i}|^2$
  if change of $error \leq$ threshold **then**
    Break
  end if
end loop
return M

Algorithm C.12 Finding inner centre point of a mesh M (see Chapter 4.5.2)

$\overrightarrow{axis}_x, \overrightarrow{axis}_y, \overrightarrow{axis}_z \leftarrow$ main axes of M (see alg. C.7)
$\overrightarrow{\mu} \leftarrow$ centre of gravity of M
$\rho \leftarrow \overrightarrow{\mu} \times (\overrightarrow{\mu} + \overrightarrow{axis}_y) \times (\overrightarrow{\mu} + \overrightarrow{axis}_z)$ {plane defined by point $\overrightarrow{\mu}$ and vectors $axis_y$ and $axis_z$}
$\boldsymbol{T} \leftarrow$ triangles of M intersected by ρ
$\boldsymbol{P} \leftarrow$ vertices of $\boldsymbol{T}$
$\overrightarrow{\mu} \leftarrow 0$
for all $\overrightarrow{p_i} in \boldsymbol{P}$ **do**
  $\overrightarrow{p_i} \leftarrow$ projection of $\overrightarrow{p_i}$ to plane ρ
  $\overrightarrow{\mu} \leftarrow \overrightarrow{\mu} + \overrightarrow{p_i}$
end for
$\overrightarrow{\mu} \leftarrow \overrightarrow{\mu}/|P|$
$\boldsymbol{I_L} \leftarrow$ intersections of polygon $\boldsymbol{P}$ and ray $-\overrightarrow{axis}_z$ from $\overrightarrow{\mu}$
$\boldsymbol{I_R} \leftarrow$ intersections of polygon $\boldsymbol{P}$ and ray $\overrightarrow{axis}_z$ from $\overrightarrow{\mu}$
if $|\boldsymbol{I_R}| mod 2 = 0$ **then** {It's still outside}
  if $|\boldsymbol{I_R}| > 0$ **then**
    $\overrightarrow{\mu} \leftarrow (\overrightarrow{I}_{R_0} + \overrightarrow{I}_{R_1})/2$
  else
    $\overrightarrow{\mu} \leftarrow (\overrightarrow{I}_{L_0} + \overrightarrow{I}_{L_1})/2$
  end if
end if
return $\overrightarrow{\mu}$

Algorithm C.13 Relaxation algorithm for parametrisation P according to [1] (see Chapter 4.5.2)

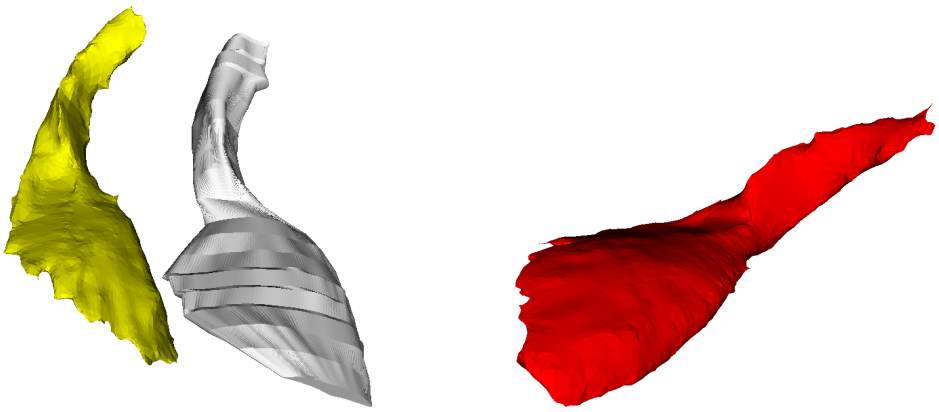
loop
  $error \leftarrow 0$
  for all triangle t_i in P **do**
    $(\overrightarrow{a}, \overrightarrow{b}, \overrightarrow{c}) \leftarrow$ vertices of t_i
    $error \leftarrow error + \overrightarrow{a} \times \overrightarrow{b} \cdot \overrightarrow{c}$
  end for
  if $error \leq$ THRESHOLD **then**
    return P
  end if
  for all vertex $\overrightarrow{p_i}$ in P **do**
    $\overrightarrow{s} \leftarrow (0, 0, 0)$
    $\boldsymbol{N_i} \leftarrow$ neighbours of $\overrightarrow{p_i}$
    for all $\overrightarrow{v_j} \in \boldsymbol{N_i}$ **do**
      $\overrightarrow{s} \leftarrow \overrightarrow{s} + (\overrightarrow{v_j} - \overrightarrow{p_i}) \cdot (\overrightarrow{v_j} - \overrightarrow{p_i})$
    end for
    $c \leftarrow 1/\max|\overrightarrow{v_j} - \overrightarrow{p_i}|$
    $\overrightarrow{s} \leftarrow \overrightarrow{s} \cdot c/|\boldsymbol{N_i}|$
    $\overrightarrow{p_i} \leftarrow \overrightarrow{p_i} - \overrightarrow{s}$
    $\overrightarrow{p_i} \leftarrow \overrightarrow{p_i}/|\overrightarrow{p_i}|$
  end for
end loop
return P

Algorithm C.14 Distribution of shift vectors $\boldsymbol{V}$ for feature points $\boldsymbol{FP}$ over spherical parametrisation P (see Chapter 4.5.4)

Require: $|\boldsymbol{FP}| = |\boldsymbol{V}|$
  for all vertex $\overrightarrow{p_i}$ in P **do**
    $W \leftarrow 0$
    for all shift vector $\overrightarrow{fp_j}$ in $\boldsymbol{FP}$ **do**
      $d \leftarrow$ angle between $\overrightarrow{p_i}$ and $\overrightarrow{fp_j}$ with respect to rotation centre $(0, 0, 0)$
      $d \leftarrow |d|/(\pi \cdot d_{max})$ {Normalise half-circle to 1}
      if $d > 1$ **then**
        $w_j \leftarrow 0$ {Too far}
      else
        $w_j \leftarrow 1 - d^{1.5}$
      end if
      $W \leftarrow W + w_j$
    end for
    for all shift vector $\overrightarrow{v_j}$ in $\boldsymbol{V}$ **do** {shift vertices matching to feature points}
      if $w_j > 0$ **then**
        $w_j \leftarrow w_j/W$ {Normalise sum to 1}
        $\overrightarrow{p_i} \leftarrow \overrightarrow{p_i} + w_j \cdot \overrightarrow{v_j}$
      end if
    end for
  end for
return P

Appendix D

Pictures



(a) Partially aligned inputs (b) Output of morphing

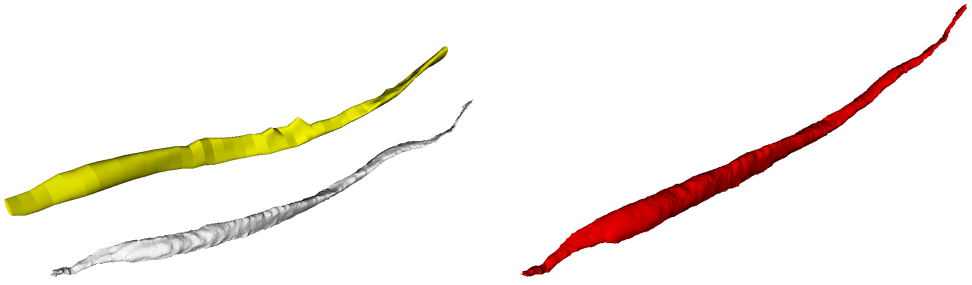
Figure D.1: Output of morphing of two manifold Iliacus muscle models.

(a) Partially aligned inputs (b) Output of morphing

Figure D.2: Output of morphing of two manifold Sartorius muscle models.

(a) Partially aligned inputs (b) Output of morphing

Figure D.3: Output of morphing of three manifold Sartorius muscle models.



(a) Partially aligned inputs (b) Output of morphing

Figure D.4: Output of morphing (red) of manifold model with 2 390 vertices (yellow) and non-manifold Sartorius model with 9 001 vertices (white).