UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE E TECNOLOGIE

A Blender plug-in for the interactive optimization of skinning in skeletal animation

Submitted 04 October 2020
Degree: **Informatica per la comunicazione digitale**

Gianluca RUBINO
910441

Supervised by Marco TARINI

Dipartimento di Informatica

Academic Year 2019/2020

Contents

List of Figures

Chapter 1

Introduction

Making an asset is a central part of 3DCG (3D computer graphics) production, typically including video-game development. Modeling animated and articulated 3D models (such as characters) is one of the most challenging tasks, requiring to address several aspects of the resulting object.

In fact, it is necessary to produce assets for the the 3D shape (i.e. controlling solely its visual appearance) and for the deformations that induce the animations to the 3D shape (i.e. controlling its movement).

Sculpting and skinning are part of the asset creation pipeline in 3DCG. We explore this more in depth in Chapter 2.

Sculpting is part of modeling. By definition, it comes before animation. It is usually used until the 3D model's appearance is completed, so it is not necessary anymore in the later stages of the pipeline.

Skinning, instead, is part of rigging. It is referred to as the act defining how a skeleton will deform the object it is bounded to. In a video-game creation pipeline, one skeleton is often bounded to many static models and it has many different animations associated to (Section 2.4). As a consequence, many 3D objects have the same multiple animations.

Skin-weight editing is relatively low-level because the artist (most often the rigger) directly modifies the weights for each bone; the process is more technical and less artistic. We try to take a different approach and modify the skinning information of a mesh through the use of sculpting tools. We directly implement this technique in a Blender plug-in.

We refer to this new approach as “Animation modeling”. It could hypothetically be a new interesting way for artists to edit animated objects on a higher-level; this hides the less intuitive transformations which happen during skeletal animation.

For example, if different dogs have the same walking and running animations, it may be necessary to change a particular mesh to a particular pose (i.e. a walk pose of a specific dog); this change will influence the rest of the animations associated to that mesh. This is already possible with skin-weight painting. Nevertheless, “animation modeling” could be a more intuitive alternative.

Chapter 2

The asset creation pipeline

We now collocate sculpting and editing of bone weights in a typical 3D workflow. Provided that the 3D asset creation pipeline is different based on the studio production or the object's properties, there are some common phases [2] (Figure 2.1).

We usually have: concept art (section 2.1), modeling (section 2.2), texturing and shading (section 2.3), rigging and animation (section 2.4). Our research focuses on video-game applications: all the assets produces in these phases are then inserted into a game engine, where they have to be combined and displayed at run-time. We briefly explain this in section 2.5.

We shall focus on the areas of the pipeline which are useful to this project: sculpting (section 2.2.1) and skinning (section 2.4.1).

2.1 Concept art

Concept art is usually part of the pre-production of an asset. It is used to previsualise the idea and the mood of the game world.

A single concept is a drawing made by a specialised artist (called concept artist) and it is often achieved with digital painting. Other tools can be used to speed-up the workflow, like: photo-bashing, kit-bashing and using real-life pictures or 3D renders as a base to draw upon.

The final concept is then passed to a 3D modeler that has to make a mesh based upon the 2D drawing. For this reason, the reference drawing should require the modeler the minimum effort to translate it from 2D to 3D space (example of a final

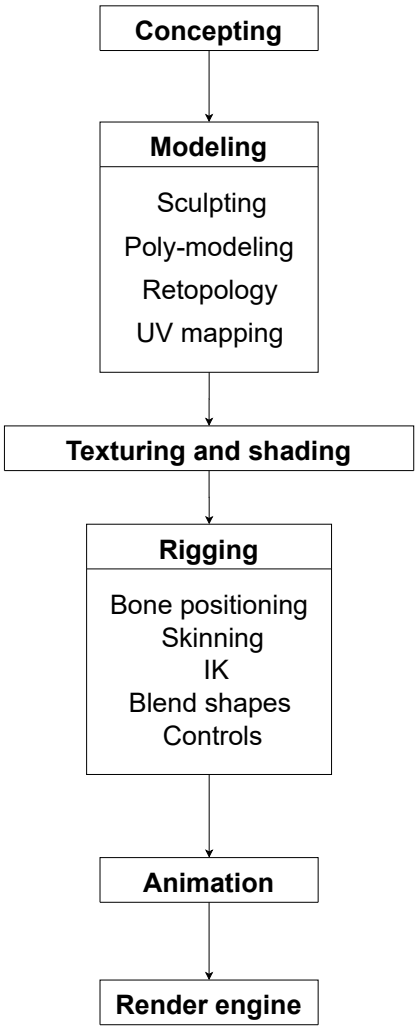

Figure 2.1: A typical video-game asset creation pipeline

concept in figure 2.2).

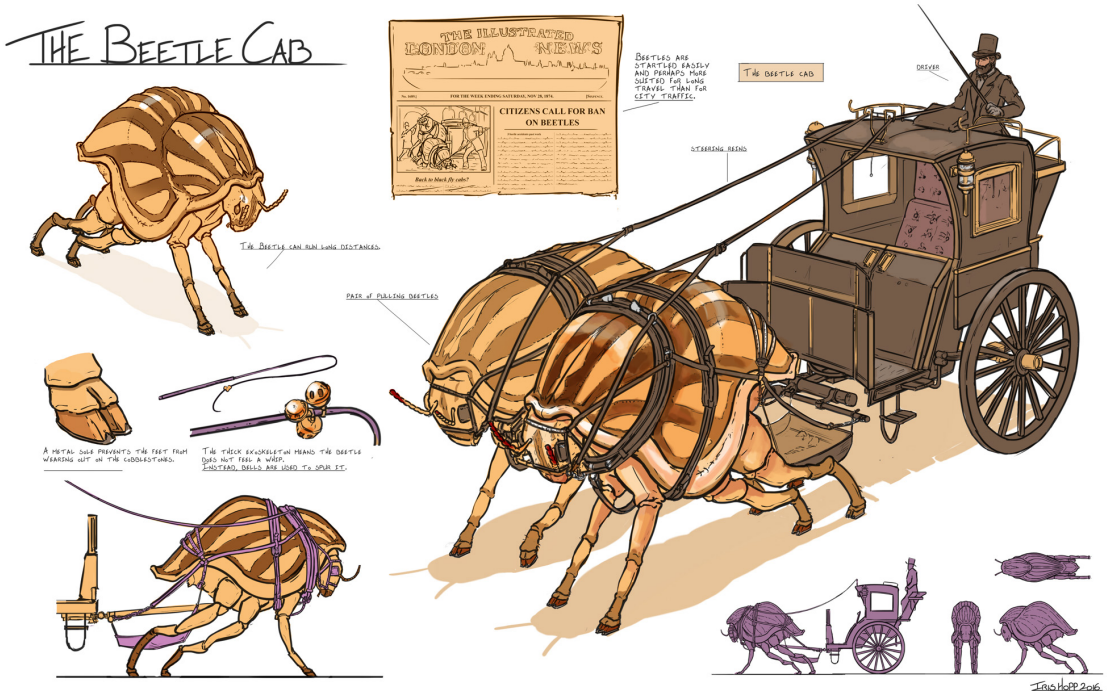

Figure 2.2: A final concept art of a fantasy carriage
Image from: Iris Hopp (Wikimedia Commons)

2.2 Modeling

A 3D model is a set of vertices in 3D space that are connected to each other in order to generate the edges and the faces that shape an object. The mesh can be thought as a net that wraps the shape of the object, while topology determine how the vertices themselves are connected.

A mesh with few polygons is called low-poly and it is easy to edit by moving each single vertex individually (hence, low-poly modelling, Figure 2.3). When a mesh has many polygons, it is called high-poly. A high-poly model is difficult to edit with low-poly modeling, unless using supporting tools or approaches like splines, modifiers or "subdiv". On the other hand, it can be molded with digital brushes (process known as sculpting, Figure 2.4).

Hi-poly models, in turn, can be organic if the silhouette is mostly composed of soft edges (for example humans, animals and fabrics), and hard surface if the silhouette is mostly composed of hard edges (for example mechanical objects).

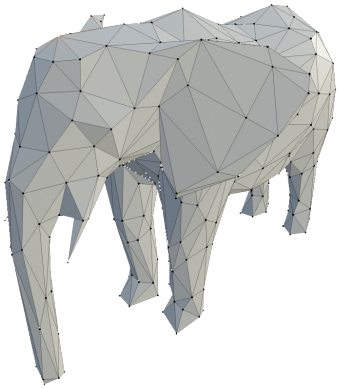

Figure 2.3: Example of low-poly modeling

Image from: ozgurcoteli (Turbosquid)

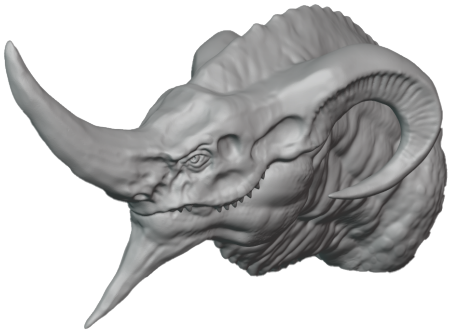

Figure 2.4: Example of a sculpted model

Image from: chauvanla (Turbosquid)

2.2.1 Sculpting

Sculpting is a method to organically model a mesh like digital clay. In fact, the skill-set required in order to achieve professional results is similar to the one used by sculptors who employ traditional mediums.

An artist has at his disposal a set of brushes to alter the mesh displacement and connectivity. Altering mesh displacement is used to build or remove form to an object in order to get it to the desired shape, whereas altering the connectivity is used in order to increase or decrease the mesh density. Increasing the vertices' number is useful to get finer details in the sculpt where needed, while decreasing the number is useful to tune its broader shape (Figure 2.5) (more details in section 3.2).

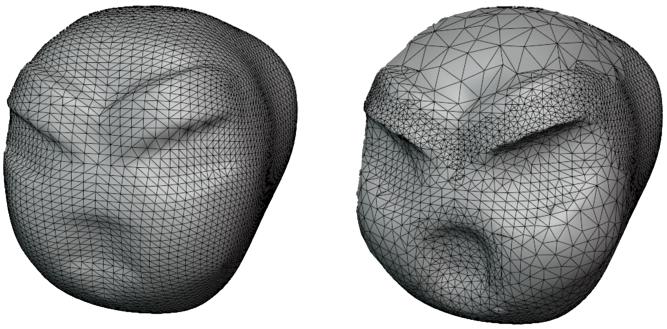

Figure 2.5: Example of a model with dynamic adaptive resolution (before/after)

2.2.2 Making a pipeline-ready model

The final steps of modeling are retopology and UV mapping.

Retopology is the action of re-making the model topology following some rules to optimise it for the pipeline. This implies that a finished model, depending on its destination, requires a certain type of topology. The ideal mesh is composed of quads, with edge loops that correctly follow its shape.

All the steps in the pipeline are optimised for quad-dominant topology. For example, game engines only use triangles, but the triangulation is implicit once the model is imported inside. UV maps, rigging and animation too are greatly favored by said topology.

Low-poly modeling allows an artist to control the shape of an object together

with its topology. The resulting mesh is ready to proceed to the next steps.

Sculpting, on the other hand, in particular if using dynamic adaptive resolution, outputs a mesh with uneven topology (example of uneven topology in a character, Figure 2.6, later retopoligized, Figure 2.7). This implies that retopology is required.

UV maps are made to map a 2D picture (called texture) on the 3D shape (more details in section 2.3). They are a 2D representation of the model's topology (an example on Figures 2.7, 2.8 and 2.10).

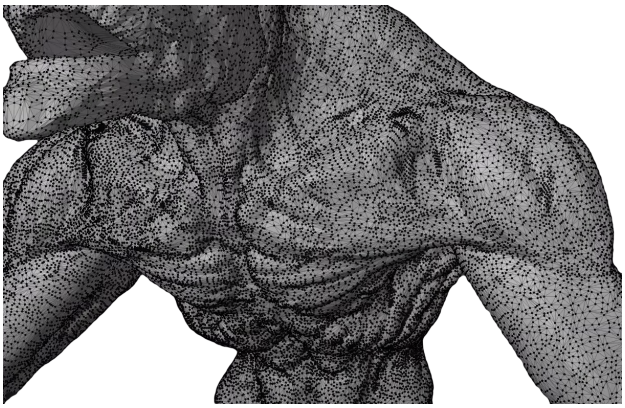

Figure 2.6: Mesh with uneven topology (after sculpting)
Image from: FlippedNormals

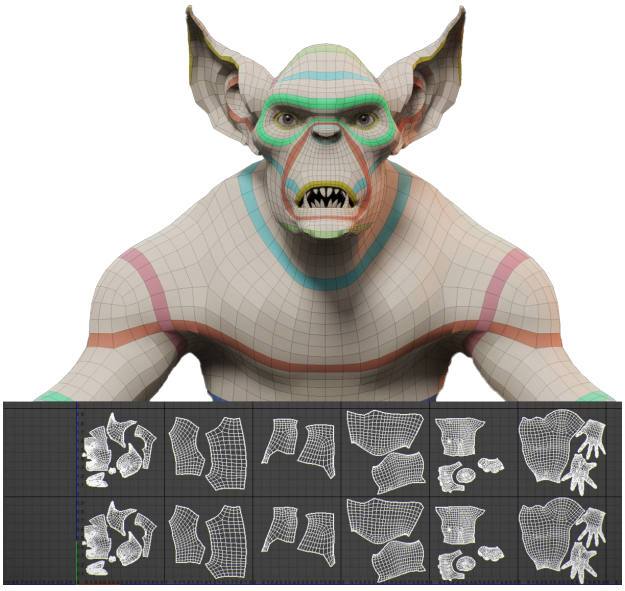

Figure 2.7: Mesh with clean topology and UV-maps
Image from: FlippedNormals

2.3 Texturing and Shading

A texture is a combination of texture elements (hence, texel or texture pixel). It is a digital image used to wrap a 3D model. They are usually a set of maps that are used to render the object.

The most common is the color map (or albedo) by which an artist paints or photo-stitches the color appearance of an object (Figure 2.9). Other maps are then used depending on the needs.

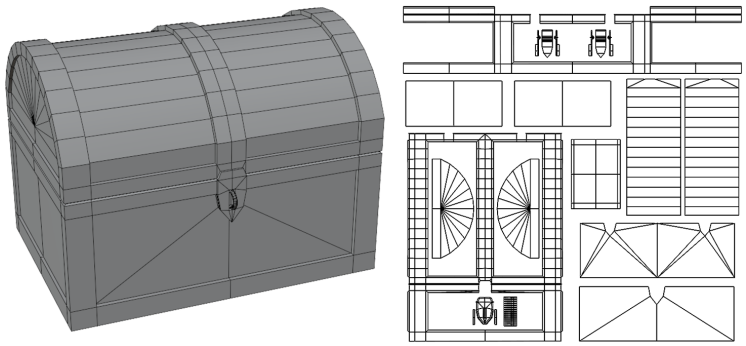

Figure 2.8: A chest with topology and UV maps

Figure 2.9: The chest rendered with textures

Furthermore, a material, also known as shader, is added in order to decide how the model will interact with the light. It is composed of parameters that set the color, glossiness, transparency, luminosity, translucency of an object. Texture maps for a model are combined to one another, inside its shader before rendering.

Shading information (computed during rendering) can be stored inside textures. This process is called baking; it optimises the calculations in expense of higher memory space.

2.4 Rigging and animation

Certain objects, for example characters, also need to be animated in order to have them move. In this case, the mesh needs to be rigged first. Rigging is the phase where the object's mesh is bound to a skeleton that drives its movement. The skeleton is composed of bones that affect the mesh vertices. Controls are then added to drive the skeleton and let the animator deform the object more easily (Figure 2.11).

Mesh vertices are influenced by one or more bones to allow more organic deformations. Skinning is the action to create or modify these influences (called "weights") in order to get to a desired deformation.

During the animation phase, an artist deforms the object in multiple poses that are then put accordingly in the timeline in order to give the illusion of movement. Each frame in the timeline which has an associated pose is called "key-frame".

Inside games, unlike for films and TV, an animator creates many small animations that may be triggered in response to a player's command. All these small animations are later stitched together inside the game engine (section 2.5).

2.4.1 Skinning (editing of bone links)

Skinning is part of the rigging phase, therefore it is a more technical process. A mesh in the rigging phase usually has already been modeled, so it has pipeline-ready topology (Figure 2.10) and UV maps. The topology from here-over should never be modified as it could compromise the rest of the pipeline.

During the rigging phase, a skeleton is created to decide how an object will deform. For a simple character, the bones that make up the skeleton are intuitively positioned at the human joints. Different or more complex models require positions based upon the required deformation.

The bones will move the vertices. Every bone affects a certain number of vertices; as such, every vertex is also affected by a certain number of bones.

A common skinning method used is LBS (linear blend skinning) [11]: it consists in giving a weight that corresponds to the influence a bone has on a vertex. Therefore, a vertex has n weights corresponding to the number of bones that affect it. The sum of all the weights should be 1, to allow the interpolation of every bone

transformation. This results in clean deformations, with limitations [5], [6].

Figure 2.10: Example of a model with clean rigging-ready topology
Image from: Demeter Dzadik (Blender Cloud)

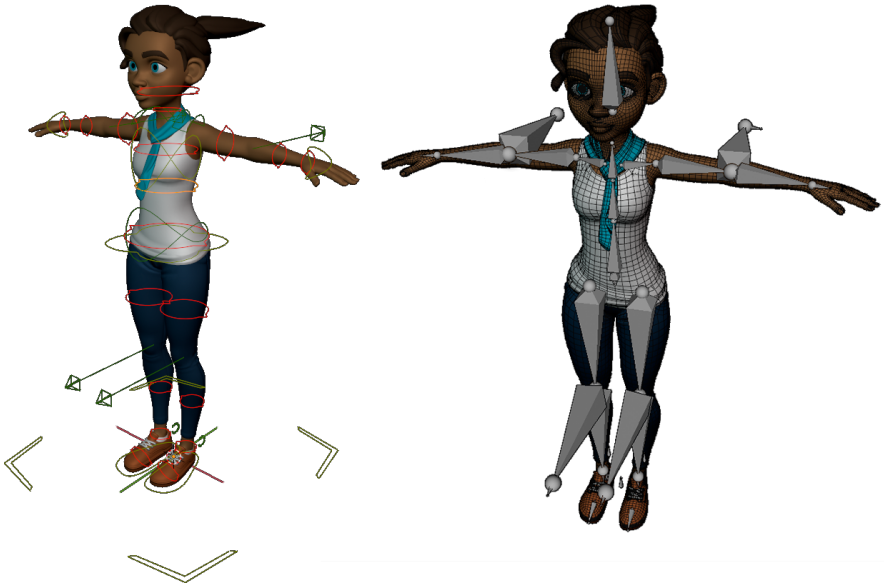

Figure 2.11: The same model with controls (left) and the skeleton that drives the motion (right)
Image from: Demeter Dzadik (Blender Cloud)

In video-games, animated objects usually have vertices affected from two to a maximum of four bones.

2.5 Game engine

Inside a game engine, each asset (2D and 3D assets, lighting and physics simulations, AI and audio) is combined in real-time. They must be merged by the game code in such a way as to create a seamless, interactive user experience. We can see how the assets produced in the pipeline on Figure 2.1 are kept separate, and stitched together inside the engine on Figure 2.12.

The animations are combined in a blend-tree where they do not have to just look good, but they also have to work alongside other data.

Moreover, game engines use real-time rendering that allows quick and smooth interaction that requires to render multiple frames every second (usually 30, 60 fps). However, obtaining physically accurate results is more complex. This is due to the algorithm involved, rasterisation (GPU-based), that doesn't try to replicate real life and, instead, requires more intervention to mimic the appearance of a scene. This is different from ray-tracing (CPU-based), used in off-line rendering for film and TV, that takes more time (seconds, minutes or hours, depending on the volume of the scene) in order to compute a single frame by approximating real-life light rays, although they are parallelizable on GPUs [7].

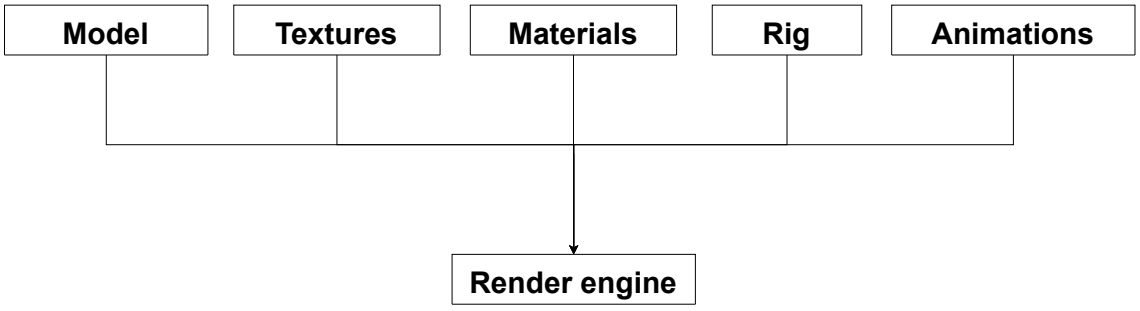

Figure 2.12: Combining the object's assets inside a game engine

Chapter 3

Background

As we discussed on Chapter 1, we design a tool that lets a user edit skin weights by using a sculpting interface. Internally, this tool solves an inverse problem: converting the 3D sculpts into skin.

In order to implement this idea, we will leverage on three existing software tools: classic skin weight editor (section 3.1), classic sculpting interfaces (section 3.2), and standard numerical solvers (section 3.3).

In this chapter, we shall discuss these three aspects of the work, that will be combined later in Chapter 7.

3.1 Skin weight editor

Authoring of skin weights is a technical skill of a digital artist (section 2.4). It is considered a challenging task to be done well and it requires a lot of trial and error to obtain good results, even for the more experienced artists [6]. This is due to the effects of the weights, i.e. their quality, that can only be evaluated by watching the deformations that they induce on each pose and animation. Observing only one pose is not enough to see the effect. The artist is also responsible of choosing the right weight amount to find the correct deformation. Vertices, in fact, often are very sensible to a slight change in influence.

Many 3D software suites used in the industry have tools to edit said weights (skin weight painting). Inside Maya and Blender, the artist paints the bones influences on the vertices, like colors. Usually, a color corresponds to a weight of 1, another

to a weight of 0 and the in-betweens have colors to blend the the first two that correspond to more or less influence (figures 3.1, 3.2). The system has the option to ensure the normalisation of the weights, depending on the pipeline workflow.

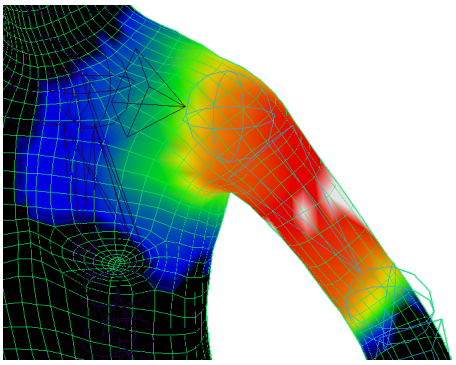

Figure 3.1: Skin editing heat-map of a left shoulder joint inside Maya

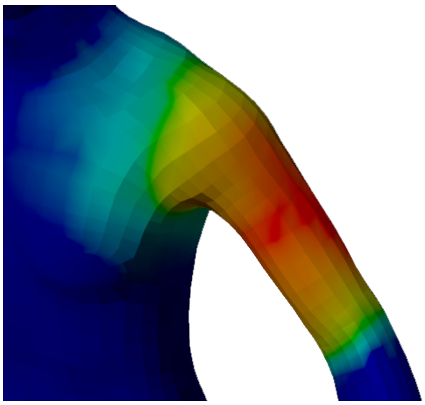

Figure 3.2: Skin editing heat-map of a left shoulder joint inside Blender

Automatic methods to calculate the weights are common: it is still an active research field. Researches mostly try to eliminate the process of manual skin authoring, or are focused on a specific area of anatomy [5].

Software suites, perhaps, offer simpler methods [4], generally used by an artist to have a base upon which edit. However, they usually do not reach a polished production-ready quality. As a consequence, manual authoring methods (like skin weight painting) still remain the most used, even though very time consuming.

The method we propose aims to simplify the assessment of any deformation by watching many poses, and to simplify the task of selecting the correct weights. As a result, it may require less trial and error.

3.2 Sculpting interfaces

Sculpting, as we introduced in section 2.2.1, is used to create static shapes geometry that will be animated in a separate step; so typical sculpting interfaces are different than our approach. Sculpting is very common, because it is intuitive for an artist, who has the illusion of manipulating plastic or clay material. He can also edit without worrying about technical details [1] like topology.

Usually the process [10] can be broken down in: blocking phase, or low frequency details, where the artist shapes a relatively low-poly mesh to get the "big forms" and proportions of an object; mid-frequency details, or secondary forms, where the object gets important landmarks: muscles, carvings. This is where a user starts to recognise the final result. In conclusion, we get high-frequency details, where the sculpt is finalised: this can include details like cloth texture or skin pores. Each phase is approximately associated to a level of details, or of subdivision. The more details are required, the more a mesh is subdivided, therefore has more polygons.

Many software suites adopt this editing mode. ZBrush, for example, is a program that focuses on sculpting (interface on figure 3.3). It has become, as of 2020, the industry standard. It provides a very smooth interaction with the object and it can support, without hassles, a mesh with up to millions of polygons. It also provides a variety of brushes.

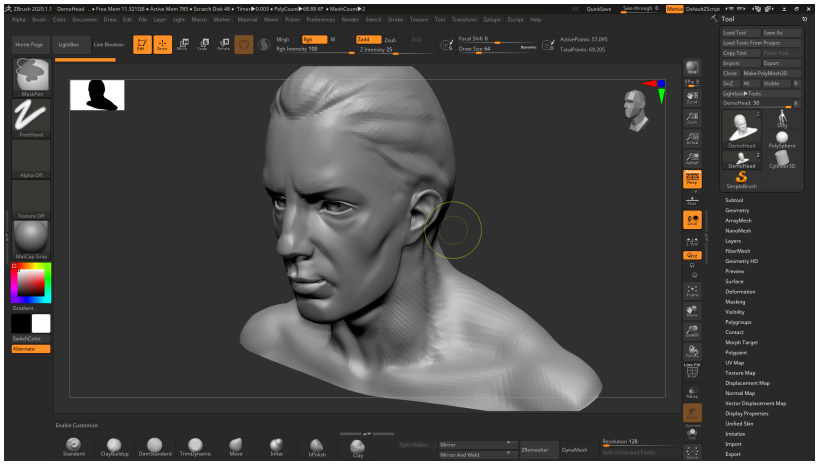

Figure 3.3: ZBrush interface

Blender also has a good sculpting mode (interface on figure 3.4). It has all the features described above, but lacks in supporting a very high polygon count. This

can be expected by the fact that the program is general purpose (it is possible to complete every task in the pipeline) and open-source.

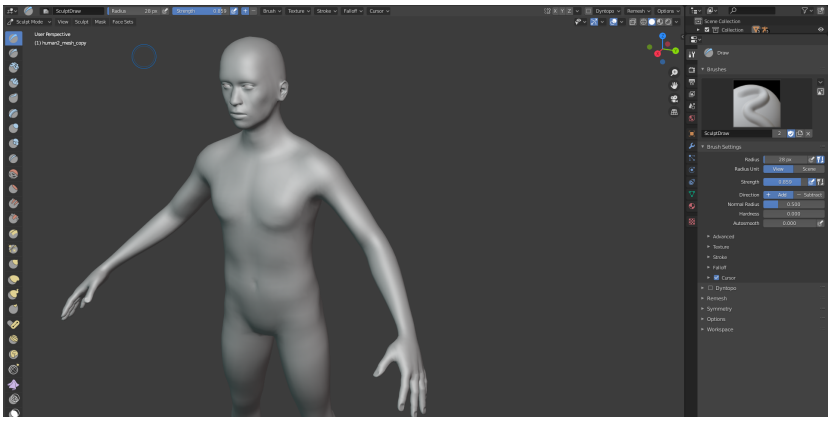

Figure 3.4: Blender sculpting interface

3.3 Numerical solvers

Our issue requires solving an inverse problem with systems of equations. In order to compute the solution, we use algorithms in numerical analysis. We can use linear least squares, or regularized least squares.

We list some notable applications intended for use with numerical analysis that may be useful for our purpose.

These are Python packages that are used for scientific computing: NumPy, SciPy, SymPy. The NumPy array is the basic data structure that many libraries understand and, together with SciPy, they provide extensive applications in numerical analysis; while SymPy is mostly used for symbolic manipulations. The Python tool stack does not attempt to replace the many critical codes and algorithms, but wraps them while providing rich data structures and programming paradigms [9].

Gurobi is a mathematical solver available for many modeling and programming languages.

Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms.

MATLAB is an environment for multi-paradigm programming language and numerical computing. It is very famous, because, as of 2020, it has more than 4 million users [8].

Pardiso is a software for solving large sparse symmetric and unsymmetric linear systems of equations on shared-memory and distributed-memory multiprocessors.

Chapter 4

Objectives

We aim to build a new tool to author the mesh skinning information through sculpting. Converting the sculpting information to vertex weights essentially means transferring the position of a vertex: this consists in a transfer from the more intuitive information of a point in 3D-space (xyz) relative to its origin, to the one of a point that is not moved by a brush, but "non-destructively" by a combination of weighted affine matrices (one per bone).

This means that, once we know the new displaced position of a vertex after a brush-stroke, we have to express it in terms of the weights that determine the transformation. The result should normally make the vertex deform in a position close to the one which has been displaced.

Every rigged model has a different number of bones that affects each vertex. In video-games, as we said in section 2.4.1, said number of bones is usually two or four. It is useful for us to have the highest possible degree of freedom, in order to get as close as possible to our objective. Consequently, we will assume that every vertex is affected by a number of four bones. We have to prepare the model for the conversion: in section 6.3 we look for a solution.

Having four bones (so four weighted matrices) gives us a certain degree of freedom. However, we will have to deal with the known limitations of matrix interpolation in skeletal animation [5], that will probably be responsible for many converted vertices in a position far from the intended one.

Chapter 5

Analysis

Now, we shall lay down the basic requirements for the software in order to support our script.

Then, we'll explain the paths to be explored before jumping to the conclusion of using Blender: either building a framework to develop the tool, or making a plug-in on an existing 3D software, in order to evaluate the applicability of our new method.

5.1 Requirements

5.1.1 Import/export assets

First, we need a software which can import and export a 3D asset in standard formats that supports mesh and animation data.

5.1.2 GUI to sculpt and to visualize animation

The software for our research needs a 3D-view and a graphical user interface (GUI) to allow a model to be inspected. Then, it needs some features in order to be able to sculpt. We only need brushes that change the displacement of vertices, because it is not necessary to change the connectivity in any way, since the topology in the skinning phase has to remain the same, as we discussed earlier (section 2.4.1).

Moreover, the program must preview a skeletal animation. This means that we should be able to inspect an imported rigged character, which also has embedded poses.

Finally, we should be able to sculpt a model while it is already in a pose.

5.1.3 Numerical optimiser

In section 3.3 we have listed certain software which might be useful to our project. Since we use Python scripting for Blender, it is natural to integrate it with NumPy and SciPy. However, they may be unnecessary: in fact, if we have a system with just three variables, inverting matrices should be sufficient (see section 6.4).

5.2 The alternatives

Summarising the options: we either write a 3D software from scratch, or we look for the applicability of our new method on an already existing one.

5.2.1 Starting from scratch

In this first option, we can use C++ as the programming language, with Qt as the IDE; then the OpenGL library for rendering. We can also add Assimp library to handle the import and export of assets.

This would give us complete control over the new program and let us develop the research knowing all its parts. On the other hand, it would require to re-write software that is already in use in many 3D suites.

5.2.2 Plug-in for an existing 3D-software

In this second option, the path allows to immediately start focusing on an implementation. However, we should perform some tests in order to check if a program has all the requirements to further develop our "animation sculpting" approach.

Considered software alternatives

It is possible to choose between many programs that can be scripted. We have considered the ones that are used more frequently in the industry, and are also common in freelance work. Therefore, we suppose they should have excellent support

for scripting: Zbrush, Autodesk Maya and Blender. Now, let us analyse each of these three suites.

Zbrush has all the necessary features for sculpting, but doesn't support skeletal animations. It has Zscript as the internal language.

Autodesk Maya is an all-purpose 3D-program that has full support for animation, but does not have efficient sculpting tools. It supports Python and MEL (Maya Embedded Language) for scripting.

Blender is a free and open-source all-purpose 3D-software that supports both animation and sculpting. It provides an extensive Python library that accesses almost all the data structures inside. Also, the source code is available to be consulted and modified.

Blender as a test framework

Blender has all the requirements to directly evaluate the applicability of our approach:

- Blender can import and export assets in common file formats;
- both animation and sculpting are well supported;
- we can import Python libraries, like NumPy, for numerical optimisation;
- the Blender Python API has a built-in operator to perform on mouse-up processing (section 7.3);
- it allows to access the source-code in order to perform changes to the program. This might be useful for our purposes.

In Chapter 6, we are going to analyse the problem code-agnostic. In Chapter 7 we'll implement the methods in Blender and we'll evaluate the results in order to see if the program is ready to further elaborate the research.

Chapter 6

Method

6.1 Overview

The user should be able to import a rigged model in the program, preview the model's poses and begin the sculpting session on a desired pose. Once the optimiser plug-in is run, skin weight desparsification (see section 6.3) is run once. The optimisation (section 6.4) is computed after each brushstroke: this only happens for the vertices which changed their position. The user can edit as much as he needs. The resulting object can, in the end, be exported with the new weights.

6.2 Computing while sculpting

The conversion process (sculpting to weights) should be done "on mouse-up", to remove the hassle of the wait of computing the conversion. This allows more forgiving computation time spans, which favors both the programmer and the artist.

In other words, when the user presses the left mouse button (or the surface with a pen tablet), every change on the mesh is registered as displacement, it is still common sculpting. As soon as the user releases the button, the conversion process begins, and it is possible to view the results immediately. The computation time should be forgiving enough to give the impression of real-time animation sculpting.

6.3 Pre-processing: skin weight desparsification

We assumed on Chapter 4 that every vertex in a mesh can be influenced by maximum four bones; as a consequence, we should write a program that (no matter what rigged mesh has in input) produces a mesh which is ready for the next step, see section 6.4.

In the original input, each vertex is linked to at most K bones. Instead, we want to identify a set of exactly N bones (four, in our case) that affect each vertex, by adding a small set of appropriate bones.

That is, we need to identify additional N-K new bones for each vertex, which originally used to be linked to just K bones. This will give more freedom to all subsequent optimizations of weights, without infringing the hard-coded constraints of computer animations used in games (see section 2.4.1). We call this process skin weight desparsification.

We select N-K additional bones, looking for the ones that are close, in the skeleton hierarchy, to the already-used K bounded: so, we use the notion of hop-distance in a graph. The new assigned bone weights will be initialised to 0. The optimiser, in the next step, will be able to assign a weight > 0 only to those new bones.

For more details on the implementation see section 7.5.

6.4 Skin weight solving

We have 4 variables, one linearly dependent from the remaining three. Thus, we can solve the system with least squares (section 6.4.2). After an in-depth analysis of the problem, it occurred that it can be divided in a series of little independent optimising problems, one per vertex (section 6.4.3); each of which can be expressed with least squares of only three variables.

As a result, it was possible to implement the solver "manually" with a library that handles matrix inversion of dimension up to 4 (section 7.6).

6.4.1 Definitions

Now, we have to define a nomenclature to illustrate the problem:
$A \in \{\mathrm{M}, \mathbf{v}, k\}$ can either be a matrix M, a vector $\mathbf{v}$ or a scalar k.

In A^c, where $c \in \{r, p\}$, r represents a position in rest pose and p represents a position in a specific pose.

In A_i, where $i \in \mathbb{N}$, i is the index of a mesh vertex. For example, $\mathbf{v}_i^c = \{x, y, z\}$ is a generic vertex of a mesh, with $x, y, z \in \mathbb{R}$.

In $A[b_j]$, where $b, j \in \mathbb{N}$, b is the index of a bone in a skeleton and j is his subscript index since a bone index is sparse. For example $\mathrm{M}[1]_5$ means that the vertex 5 is associated to the bone matrix 1.

6.4.2 Least squares method

Having N vertices and M bones in a mesh.

Knowing that we are in sculpt mode with our desired mesh selected. For each vertex $k \in \{0, \ \ldots \ , N-1\}$ that has been displaced by a brush stroke, we do the following.

Having $\mathbf{v}_k^r$ the starting vertex position in a mesh in rest pose space and $\mathbf{t}_k^r$ the vertex in rest pose displaced after a brush stroke in sculpt mode. A vertex has 4 starting weights $\{w[a_0], w[a_1], w[a_2], w[a_3]\}_k$, with $a \in \{0, \ \ldots \ , M-1\}$ relative to 4 rest pose bones matrices $\mathrm{M}[a_i]_k^r$ and 4 pose-specific bone matrices $\mathrm{M}[a_i]_k^p$, with $i \in \{0, 1, 2, 3\}$.

The matrix transformation for each bone is:

$$\mathrm{M}[a_i]_k = \mathrm{M}[a_i]_k^p (\mathrm{M}[a_i]_k^r)^{-1}$$

We can obtain the transformation matrix to take a vertex from rest-pose space to pose-specific space:

$$\mathrm{T}_k = \sum_{i=0}^{3} w[a_i] \mathrm{M}[a_i]_k$$

Now we can obtain the displaced vertex in pose-specific space:

$$\mathbf{t}_k^p = \mathrm{T}_k \mathbf{t}_k^r$$

The equation we are trying to solve, with $w[a_i]^f$ the final weights we are looking for:

$$\sum_{i=0}^{3} w[a_i]^f (\mathrm{M}[a_i]_k \mathbf{v}_k^r) = \mathbf{t}_k^p$$

By setting $\mathbf{d}_i = \mathrm{M}[a_i]_k \mathbf{v}_k^r$ and knowing that $w[a_3] = 1 - (w[a_0] + w[a_1] + w[a_2])$, we get:

$$w_0^f(\mathbf{d}_0 - \mathbf{d}_3) + w_1^f(\mathbf{d}_1 - \mathbf{d}_3) + w_2^f(\mathbf{d}_2 - \mathbf{d}_3) = \mathbf{t}^p - \mathbf{d}_3$$

We can order the equation as matrices:

$$\mathrm{D} = \begin{bmatrix} (\mathbf{d}_0 - \mathbf{d}_3)_x & (\mathbf{d}_1 - \mathbf{d}_3)_x & (\mathbf{d}_2 - \mathbf{d}_3)_x \\ (\mathbf{d}_0 - \mathbf{d}_3)_y & (\mathbf{d}_1 - \mathbf{d}_3)_y & (\mathbf{d}_2 - \mathbf{d}_3)_y \\ (\mathbf{d}_0 - \mathbf{d}_3)_z & (\mathbf{d}_1 - \mathbf{d}_3)_z & (\mathbf{d}_2 - \mathbf{d}_3)_z \end{bmatrix} \quad \mathbf{w} = \begin{bmatrix} w[a_0]^f & w[a_1]^f & w[a_2]^f \end{bmatrix}$$

so

$$\mathbf{w} = \mathrm{D}^{-1}(V_t - d_3)$$

And the final weights should be:

$$\mathbf{w}_k^f = \{w[a_0]^f, w[a_1]^f, w[a_2]^f, w[a_3]^f\}$$

with

$$w[a_3]^f = 1 - (w[a_0]^f + w[a_1]^f + w[a_2]^f)$$

6.4.3 Regularised least squares method

The approach in section 6.4.2 does not work on its own, because we cannot infer a position by finding the minimiser based just upon the target position. As a result, the system is ill-posed, therefore it is impossible to fit and most of the matrices we construct do not have an inverse.

We have to take into consideration the starting weights (or old weights) of each vertex, in order to tell the system that the "new" weights we are looking for should

tend to the initial state. In this way we obtain a RLS system that allows the introduction of further constraints that uniquely determine the solution.

By having:

$$\mathrm{A}\mathbf{w} = \mathbf{b} \tag{6.1}$$

as the generalised equation. A is a 3x3 matrix, $\mathbf{b}$ is the 3D column vector of the known terms and $\mathbf{w}$ is the column vector of the three independent variables. We want to find:

$$\min \mathbf{w} \|\mathrm{A}\mathbf{w} - \mathbf{b}\|^2 \Rightarrow f'(\mathbf{w}) = 0$$

$$f(\mathbf{w}) = \|\mathrm{A}\mathbf{w} - \mathbf{b}\|^2 = \mathbf{w}^T \mathrm{A}^T \mathrm{A}\mathbf{w} - 2\mathbf{w}^T \mathrm{A}^T \mathbf{b} + \mathbf{b}^T \mathbf{b}$$

$$f'(\mathbf{w}) = 2\mathrm{A}^T \mathrm{A}\mathbf{w} - 2\mathrm{A}^T \mathbf{b}$$

$$f'(\mathbf{w}) = 0 \Rightarrow 2\mathrm{A}^T \mathrm{A}\mathbf{w} = 2\mathrm{A}^T \mathbf{b}$$

$$\mathrm{A}\mathbf{w} = \mathbf{b}$$

$$\min \mathbf{w} = \mathrm{A}^{-1}\mathbf{b}$$

Setting $\mathbf{b}' = -2\mathbf{b}$

$$\min \mathbf{w} = -\frac{1}{2}\mathrm{A}^{-1}\mathbf{b}'$$

This is possible, for our problem, only by using RLS on equation 6.1. This means we have to compose multiple Quadratics in order to obtain the final system.

The first one is, seen previously in section 6.4.2:

$$\|\mathrm{D}\mathbf{w} - \mathbf{p}\|^2 = \mathbf{w}^T \mathrm{D}^T \mathrm{D}\mathbf{w} - 2\mathbf{w}^T \mathrm{D}^T \mathbf{p} + \mathbf{p}^T \mathbf{p}$$

where $\mathbf{p} = \mathbf{t}^p - \mathbf{d}_3$.

Having the starting, known, weights for a vertex:

$$\mathbf{o}_k = \{o[a_0], o[a_1], o[a_2], o[a_3]\}$$

We construct the second Quadratic that regularises the new weights based upon the first three starting ones. The parameter k will control the power of the regularisation, therefore also the invertibility of the matrix.

$$\mathbf{o}^{3d} = \{o[a_0], o[a_1], o[a_2]\}$$

$$\|k\mathrm{I}\mathbf{w} - k\mathbf{o}^{3d}\|^2 = \mathbf{w}^T(k^2\mathrm{I})\mathbf{w} - 2\mathbf{w}^T k^2(\mathbf{o}^{3d})^T + k^2(\mathbf{o}^{3d})^T\mathbf{o}^{3d}$$

We now construct the third Quadratic that regularises the new weights based upon the fourth weight $o[a_3]$.

Having J_3 a $3x3$ Matrix of all ones and $\mathbf{j} = \mathrm{J}_{3,1}$ a column Vector of all ones.

$$\|k\mathrm{J}_3\mathbf{w} - k(o[a_3]\mathbf{j} - \mathbf{j})\|^2 = \mathbf{w}^T(k^2\mathrm{J}_3)\mathbf{w} + 2\mathbf{w}^T k^2(o[a_3]\mathbf{j} - \mathbf{j}) + k^2(o[a_3] - 1)^2$$

We can now sum all three elements to arrive to:

$$\mathrm{A} = \mathrm{D}^T\mathrm{D} + k^2\mathrm{I} + k^2\mathrm{J}_3$$

$$\mathbf{b}' = -2\mathrm{D}^T\mathbf{p} - 2k^2(\mathbf{o}^{3d})^T + 2k^2(o[a_3]\mathbf{j} - \mathbf{j})$$

$$c = \mathbf{p}^T\mathbf{p} + k^2(\mathbf{o}^{3d})^T\mathbf{o}^{3d} + k^2(o[a_3] - 1)^2$$

So the final Quadratic will be:

$$\mathbf{w}^T\mathrm{A}\mathbf{w} + \mathbf{b}'\mathbf{w}^T + c \tag{6.2}$$

And the minimiser:

$$\mathbf{w}_k^{min} = \min \mathbf{w} = -\frac{1}{2}\mathrm{A}^{-1}\mathbf{b}'$$

6.4.4 Normalising

Our linear system does not have a weight boundary between 0 and 1. Our implementation in Blender requires this limitation. Since we are solving the system manually, and not by using dedicated libraries, we need to impose it.

We make a new Quadratic that will then be added to the previous system only if a specific weight value in the vector of minimisers is negative.

$\mathbf{w}_k^{min}$ is made of each component of the vector of minimisers also with the linearly dependent fourth component:

$$\mathbf{w}_k^{4d} = \{w[a_0], w[a_1], w[a_2], w[a_3]\}$$

We set h as the parameter that controls the power of the normalisation.

$$\mathrm{M_x} = \begin{bmatrix} h & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \mathrm{M_y} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & h & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \mathrm{M_z} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & h \end{bmatrix}$$

The new matrix is added to the A in formula (6.2) based upon which element of the vector is negative.

The fourth weight is different from the other three: the new Quadratic is still added to the system only if $w[a_3]$ is negative.

$$\mathbf{l} = \{-2, -2, -2\}$$

$$\mathbf{w}^T(\mathrm{A} + h\mathrm{J}_3)\mathbf{w} + (\mathbf{b}' + h\mathbf{l})\mathbf{w}^T + hc$$

The minimiser of the new Quadratic should ensure that the boundary $[0, 1]$ is respected.

In conclusion, in order to remove any fluctuations in machine error, we apply a crude algorithm to normalise the vectors. This algorithm takes a vector, sums each element to the opposite its minimum and then divides it to its maximum. This happens only if the minimum is negative, or the maximum is > 1. More on the implementation Chapter (section 7.6.3).

This approach, for now, does not produce good results. We can see the effects of the normalisation on a model in section 8.4.

Chapter 7

Implementation

7.1 Basics of scripting in Blender

We can access every aspect of the program with Blender's own API (Application Programming Interface) called bpy (Blender Python).

```
import bpy
```

Other libraries will be used, but they will be specified later when necessary.

We now need a way to access information about: the mesh (vertices, edges, faces), the rig (skeleton, bones, rest-pose, any other pose) and the skinning (weights per vertex and referred bone). The skeleton in Blender is called armature.

In order to obtain the object data, there first has to be a reference from the members available in the context: the area of Blender which is currently being accessed.

```
object = bpy.context.active_object
```

Once we get the correct reference, we can access its data:

```
object_data = object.data
vertices = object_data.vertices

for vertex in vertices:
    print("Coordinates xyz of a single vertex:", vertex.co)
```

The same can be performed for any other data structure, with a few exceptions. We will specify them later when needed.

In addition, we use two libraries in order to facilitate our calculation. One is NumPy (section 3.3) and the other is Mathutils, a library used in Blender [3] to manage Vectors and Matrices.

```
import numpy as np
import mathutils
```

7.2 Blender Modal Operator

Operators are a way to implement a Python script into Blender [3].

The modal operator is crucial to achieve on mouse-up processing, as it is fundamentally built to perform actions while other Blender functions are running; and it does not freeze the interface during a script computation.

We shall take a look at its main structure.

7.2.1 Inside a Class

Every new operator in Blender inherits from the Class Operator.

```
class SimpleModalOperator(bpy.types.Operator):
    def execute(self, context):
    def invoke(self, context, event):
    def modal(self, context, event):
```

The execute function is the simplest way to run commands without the user input. It also allows the operation to be repeated from a macro or another script.

The invoke function is used at the moment the operator is called, to initialize it from the current context. It is typically used to assign properties which are later used by other functions.

The modal function is the core of the class, as it will keep being run to handle events until it returns "FINISHED" or "CANCELLED". Every time a new event is detected, such as a mouse click or key press, the function will be called.

7.2.2 Setting initial parameters

Inside the invoke method, we can specify initial parameters that than can be used inside the modal method to be processed.

```
def invoke(self, context, event):
    #some useful functions...

    #add the modal operator to the window manager (handles open windows and other
    user interface data)
    context.window_manager.modal_handler_add(self)

    #the operator does not exit immediately after invoke finishes:
    return {'RUNNING_MODAL'}
```

Inside the modal method we can detect the user actions and determine when the user presses and releases the left mouse button.

```
def modal(self, context, event):
    if event.type=='LEFTMOUSE' and event.value=='PRESS':
        self.mousePressed = True
```

```
        #instructions to perform when the left mouse button is pressed

    if event.value == 'RELEASE' and self.mousePressed:
        self.mousePressed = False
        #instructions to perform when the left mouse button is released
```

Sometimes, the event value "RELEASE" pops-up twice after the mouse button is physically released, so it is necessary to use a Boolean variable to check if the mouse is already pressed.

7.3 Testing on mouse-up processing

We write a simple program that averages the distance between the position of a vertex, before and after being edited. This is a test to evaluate the ability to process the mesh after each brush-stroke. It is also useful to check where the mesh is being edited and apply a function only to those specific vertices.

```
#inside the modal method
if event.type == 'LEFTMOUSE' and event.value == 'PRESS':
    self.mousePressed= True

    #get all vertex positions
    for v in bpy.context.active_object.data.vertices:
        #startingVerts is a class variable list
        self.startingVerts.append( v.co.copy() )

if event.value == 'RELEASE' and self.mousePressed:
    self.mousePressed= False
    meshVertices = bpy.context.active_object.data.vertices

    for v in meshVertices:
        oldPosition = self.startingVerts[v.index]
        newPosition = v.co.copy()

        #apply function only to modified vertices
        if (oldPosition != newPosition):
            #average
            v.co = ( oldPosition + newPosition) / 2.0
```

We have to make sure the brush doesn't modify the connectivity, as we specified in section 5.1.2:

```
#inside the invoke method
workspTools = bpy.context.workspace.tools
pen = workspTools.from_space_view3d_mode(bpy.context.mode)
if pen.idname[0:13] == 'builtin_brush':
    #raise an error if it is the wrong brush
```

We can then specify inside the modal method when the operator can be closed intentionally by the user. In this case, for debugging purposes, we can quickly close the operator by pressing the right mouse button or the escape key.

```
#inside the modal method
if event.type in {'RIGHTMOUSE', 'ESC'}:
    return {'CANCELLED'}
```

7.4 Access to vertices, bones and weights

Blender divides its armature information into three different data blocks: vertices, bones and vertex groups.

A vertex group has stored the weight of each vertex. They are a separate data block from bones and vertices, as they are not utilized just for skinning. For example, they are also used on a mesh in order to decide where particle emitters should emanate from.

Each vertex influenced by n bones will be inside n vertex groups. This lets us easily access the weights of a vertex and its associated bone.

A bone has different indices from its vertex group, but they have the same name, that is how we can make sure we are accessing the correct data to correctly map a vertex weight to its bone.

The list of bones can be parsed both as an array-like element or a graph-tree. We can access the transformation matrix of each bone from there; we will later interpolate the matrices of each bone bounded to a vertex, with its weight. This allows to get the final transformation of each vertex from the position in rest pose to the position in a specific pose.

7.5 Skin weight desparsification

7.5.1 Hop-distance algorithm

We select the first N-K bones j that minimise $\sum i \in K w_i * distance(j, i)$ where K is set of bones i that initially have weight $w_i > 0$.

```
def proximity( j, k ):
    maximum = 0
    for i in bones:
       maximum = max (
          maximum,
          proximity_to_children( i , k ) * proximity_to_children( i , j )
       )
    return maximum;


def proximity_to_children( j , k ):
    root = bones[0]
    result = 1
    while 1:
        if j==k:
            return result

        if k == root:
            return 0
```

```
        result *= 0.5;
        k = k.parent;

#MAIN
object = bpy.context.active_object
bones = object.parent.data.bones

vertices = object.data.vertices

boneCount = len(bones)
proximities = np.zeros( (boneCount, boneCount) )

bonesIndeces = {}
for i in range(boneCount):
    bonesIndeces[bones[i].name] = i

for j in range(boneCount):
    for k in range(boneCount):
        proximities[j][k] = proximity( bones[j], bones[k] )

for v in vertices:
    bonesVert = []
    weights = []
    for w in v.groups:
        boneName = object.vertex_groups[w.group].name
        bonesVert.append(bones[boneName])
        weights.append(w.weight)

    if v.index == 4672:
        print(boneNameStrings(bonesVert))
        print(weights, "\n")

    combined_dist_per_bone = []
    for b in bones:
        proxBoneVert = 0
        if b not in bonesVert:
            for j in range( len(bonesVert) ):
                proxBoneVert += proximities[ bonesIndeces[bonesVert[j].name] ][
    bonesIndeces[b.name] ] * weights[j]
            combined_dist_per_bone.append(proxBoneVert)
        else:
            combined_dist_per_bone.append(1)

    newbonesVert = [0,0,0,0]
    maxes = [0,0,0,0]
    for i in range( boneCount ):
        if maxes[0] < combined_dist_per_bone[i]:
            maxes[0] = combined_dist_per_bone[i]
            newbonesVert[0] = bones[i]
        elif maxes[1] < combined_dist_per_bone[i]:
            maxes[1] = combined_dist_per_bone[i]
            newbonesVert[1] = bones[i]
        elif maxes[2] < combined_dist_per_bone[i]:
            maxes[2] = combined_dist_per_bone[i]
            newbonesVert[2] = bones[i]
        elif maxes[3] < combined_dist_per_bone[i]:
            maxes[3] = combined_dist_per_bone[i]
            newbonesVert[3] = bones[i]

    if v.index == 4672:
        print(boneNameStrings(newbonesVert))

    for bone in bonesVert:
        if bone not in newbonesVert:
            object.vertex_groups[bone.name].remove([v.index])

    for bone in newbonesVert:
        if bone not in bonesVert:
            object.vertex_groups[bone.name].add([v.index], 0.0, 'REPLACE')
```

7.5.2 Another algorithm in pseudo-code

Let us now introduce a further approach to the desparsification matter. We tested another algorithm, in order to evaluate how it compares to the previews one in terms of choosing the bones.

It goes as follows: We first add new weights by parsing each edge in each triangle and let each vertex on that edge "infect" one another. By influencing we mean: if the first vertex has a bone that the second one does not have, the first one "gives" its bone to the second vertex that does not have it. The weight assigned to the second vertex in equal to the distance from this new bone. In case a vertex is influenced again and finds a closer bone from another vertex, then the further away bone will be replaced with the closer one.

In other words: we interpret the influence that a new weight could have on the vertex by calculating the distance between the vertex itself and a new possible bone.

In pseudo-code:

```
def infectWeights(vertex_a, vertex_b):
    #if vertex b needs weights
    if len(vertex_b.weights) < 4:
      for each bone of vertex_a:
        #obtain the influence that a bone could have on the vertex
        distance_from_bone := vertex.pos - bone.pos
        dist := round( distance_from_bone.magnitude() )
        if bone not in vertex_b.bones:
          #assign negative distance from bone
          vertex_b.bone.weight := -dist
          something_has_been_assigned := True
          break for
        else:
          for each weight in vertex_a:
            #if there is a weight that is more relevant than the current bone
    influence
            if vertex_a.weight < dist:
              vertex_b.bone.weight := dist
              something_has_been_assigned := True
              break for

def addBoneWeights():
  something_has_been_assigned := True
  while something_has_been_assigned:
    something_has_been_assigned := False
    for each triangle:
      for each edge in the triangle:
        firstVertexIndex := edgeVertex[0]
        secondVertexIndex := edgeVertex[1]
        #first infects second
        infectWeights(firstVertexIndex, secondVertexIndex)
                #second infects first
            infectWeights(secondVertexIndex, firstVertexIndex)

#set all negative weights to zero
for each vertex:
  for boneName in vert:
    if vert[boneName] < 0.0:
      vert[boneName] = 0.0
```

Then we select all the vertices with more than four attached bones and remove

the ones that have the least influence.

In pseudo-code:

```
for each vertex:
  i := current_vertex_index

  #REMOVE excess bone weights
  if len(vertex.weights) > 4:
    weights := vertex[i].getWeights
    weights.sort("descending order")
    weights := weights["first four values"]
    vertex[i].weights := weights
```

The bone choice in this algorithm, after some tests with our placeholder rig (Figure 8.2), does not differ from the one in section 7.5.1.

Since the results are identical, in section 8.3 we show the image of only one of the algorithms.

7.6 Skin weight solving in Blender

7.6.1 Quadratic Class

This Class is created in order to manage all the Quadratics in our program. It lets us keep each quadratic object separate, and enhances the operations between them.

```
class Quadratic():
    #constructor, identify a quadratic with its three elements
    def __init__(self, A, b, c):
        self.A = A #3x3 Matrix
        self.b = b #3d Vector
        self.c = c #Scalar

        #The minimiser is set, but it is not yet calculated as it may not exist or
    it optimises performance, as it is not necessary to calculate for each new
    object
        self.w_min = None

    #to string override
    def __str__(self):
        t = "Matrx A:\n\t" + str(self.A) + "\nVector b = " + str(self.b) + " |
    scalar c = " + str(self.c)
        if self.w_min:
            t += "\nMinimiser: " + str(self.w_min)
        else:
            t += "\nMinimiser non yet calculated or matrix has no inverse."
        return t + "\n"

    #evaluate the quadratic with any vector
    def evaluate(self, w):
        return w @ self.A @ w + self.b @ w + self.c

    #get the minimiser of the quadratic and assign it to the object state
    def get_minimiser(self):
        self.w_min = (-1/2) * self.A.inverted() @ self.b

    #override the addition operation
    def __add__(self, quadratic):
        A = self.A + quadratic.A
        b = self.b + quadratic.b
        c = self.c + quadratic.c
```

```
        return Quadratic(A, b, c)

    #get the fourth weight from a vector or from the minimiser
    def to_4d_weights(self, w):
        newW = w.copy()
        newW.resize_4d()
        newW.w = 1 - (w.x + w.y + w.z)
```

7.6.2 In the code

Relevant methods

These methods are inside the Modal Operator Class and will be used in the main calculation.

```
def normalizeVector(self, vector):
    min = 0
    max = 1
    for i in range(4):
        if vector[i] < min:
            min = vector[i]
        if vector[i] > max:
            max = vector[i]

    sum = 0
    for i in range(4):
        vector[i] += -min
        vector[i] /= max - min
        sum += vector[i]

    for i in range(4):
        vector[i] /= sum

    return vector

def make_p_vector(self, final_fromRest_toPose_matrices, oldPos_inRest):
    p = []
    for FinalToPoseMatrix in final_fromRest_toPose_matrices:
        p.append(FinalToPoseMatrix @ oldPos_inRest)
    return p

#Interpolate a matrix manually with newfound weights to predict what Blender should
     do (same as interp_p_weights)
def interpolate_transform(self, vertexCoord_rest, final_fromRest_toPose_matrices,
    newW):
    final_interp_fromRest_toPose = mathutils.Matrix(( (0,0,0,0), (0,0,0,0),
    (0,0,0,0), (0,0,0,0) ))
    for j in range(4):
        final_interp_fromRest_toPose += newW[j] * final_fromRest_toPose_matrices[j]
    return final_interp_fromRest_toPose @ vertexCoord_rest

#Interpolate p vector with newfound weights to predict w hat Blender should do (
    same as interpolate_transform)
def interp_p_weights(self, p, weight):
    return weight.x * p[0] + weight.y * p[1] + weight.z * p[2] + weight.w * p[3]

#Assign newfound weights to Blender's data structure
def assignNewWeights(self, vertex, newWeights):
    j = 0
    for w in vertex.groups:
        boneName = self.meshObject.vertex_groups[w.group].name
        self.meshObject.vertex_groups[boneName].add([vertex.index], newWeights[j],
    'REPLACE')
        j += 1
```

Inside the modal method

Inside the modal method, we check when the mouse is released and we compute the new weights.

```
if event.value == 'RELEASE' and self.mousePressed:
    self.mousePressed = False

    for v in self.meshVertices:
        #get original vertex position an position after brush stroke
        oldPos_inRest = self.vertsBeforeEdit[v.index].copy()
        newPos_inRest = v.co.copy()

        #get starting bone vertex weights (old weights)
        oldWeights = self.getAssignedWeights(v)

        #get vertex bone matrices
        #get vertex pose bone matrices
        #get final transformation matrix interpolated with old weights
        global_toRest_matrices = []
        global_toPose_matrices = []
        final_fromRest_toPose_matrices = []
        final_interp_fromRest_toPose =  mathutils.Matrix(( (0,0,0,0), (0,0,0,0),
    (0,0,0,0), (0,0,0,0) ))

        for w in v.groups:
            boneName = self.meshObject.vertex_groups[w.group].name

            bone = self.armatureObject.data.bones[boneName]
            global_toRest = bone.matrix_local
            global_toRest_matrices.append( global_toRest.copy() )

            poseBone = self.armatureObject.pose.bones[boneName]
            global_toPose = poseBone.matrix
            global_toPose_matrices.append( global_toPose.copy() )

            final_fromRest_toPose = global_toPose @ ( global_toRest.inverted() )
            final_fromRest_toPose_matrices.append( final_fromRest_toPose )

            final_interp_fromRest_toPose += w.weight * final_fromRest_toPose

        #get old and new (target) position in current pose
        oldPos_inPose = final_interp_fromRest_toPose @ oldPos_inRest
        newPos_inPose = final_interp_fromRest_toPose @ newPos_inRest

        #Compute new weights to edited area
        if (newPos_inRest-oldPos_inRest).magnitude != 0:

            #resetVertexPosition
            v.co = oldPos_inRest

            #make destination vertices
            p = self.make_p_vector(final_fromRest_toPose_matrices, oldPos_inRest)

            #linear system composition
            Tmatrix = mathutils.Matrix((
                (p[0]-p[3]),
                (p[1]-p[3]),
                (p[2]-p[3])
            ))

            Tmatrix.transpose()

            target_minus_p3 = newPos_inPose - p[3]

            #solve linear system

            #QUADRATIC 0
            quadr0 = Quadratic(
                (Tmatrix.transposed() @ Tmatrix),
                (-2*Tmatrix.transposed() @ target_minus_p3),
```

```
                (target_minus_p3 @ target_minus_p3) )

            #QUADRATIC 1
            k = 0.05 #regularisation parameter
            oldW_3d = oldWeights.to_3d() #row vector 3d
            quadr1 = Quadratic(
                ( k * k * mathutils.Matrix.Identity(3) ),
                (-2 * k * k * oldW_3d),
                (k * k * oldW_3d @ oldW_3d) )

            #QUADRATIC 2
            ones = mathutils.Matrix( ((1,1,1), (1,1,1), (1,1,1)) )
            w3_3d = mathutils.Vector( ( (oldWeights.w - 1), (oldWeights.w - 1), (
    oldWeights.w - 1) ) )
            quadr2 = Quadratic(
                ( k * k * ones ),
                ( 2 * k * k * w3_3d),
                ( k * k * (oldWeights.w - 1) * (oldWeights.w - 1) ) )

            #QUADRATIC 4
            quadr3 = quadr0 + quadr1 + quadr2

            #RESTULT quadr 3
            quadr3.get_minimiser()

            #Final non normalised weights
            newW = quadr3.to_4d_weights(quadr3.w_min)

            #continue to normalisation
```

7.6.3 Normalisation

We now implement the method discussed in section 6.4.4.

```
h = 1.2 #normalisation parameter
normWeights = newW.copy()

#QUADRATIC 5
ones = mathutils.Matrix( ( (1,1,1), (1,1,1), (1,1,1) ) )
Ax = mathutils.Matrix( ( (1,0,0), (0,0,0), (0,0,0) ) )
Ay = mathutils.Matrix( ( (0,0,0), (0,1,0), (0,0,0) ) )
Az = mathutils.Matrix( ( (0,0,0), (0,0,0), (0,0,1) ) )
zeros = mathutils.Vector()

quadr5 = quadr3

#Assign normalise Quadratic to a negative weight only

#We take in consideration machine error
epsilon = 0.000000012

if normWeights.x < -epsilon:
    quadr5 += Quadratic(Ax*h, zeros, 0)
    quadr5.get_minimiser()
    normWeights = quadr5.to_4d_weights(quadr5.w_min)

if normWeights.y < -epsilon:
    quadr5 += Quadratic(Ay*h, zeros, 0)
    quadr5.get_minimiser()
    normWeights = quadr5.to_4d_weights(quadr5.w_min)

if normWeights.z < -epsilon:
    quadr5 += Quadratic(Az*h, zeros, 0)
    quadr5.get_minimiser()
    normWeights = quadr5.to_4d_weights(quadr5.w_min)

if normWeights.w < -epsilon:
    quadr5 += Quadratic(ones*h, mathutils.Vector( (-2,-2,-2) )*h, 1*h)
    quadr5.get_minimiser()
    normWeights = quadr5.to_4d_weights(quadr5.w_min)
```

```
normWeights = self.normalizeVector(normWeights)

#assign new solved weights
self.assignNewWeights(v, normWeights)
```

Chapter 8

Results

We are now going to show the results of our research and of the related analysis.

8.1 Glass and average

The first mesh used to test the script that implements the modal operator to average a brush stroke is on Figure 8.1. It is a 3D photo scan of a left over glass after manufacturing.

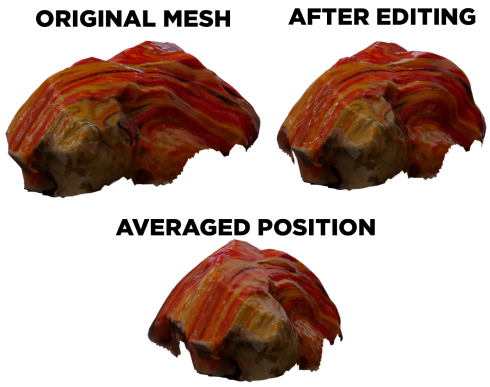

Figure 8.1: The object used to test the average

8.2 Character

The rigged character used for testing the optimisation is on figure 8.2. It was generated with the program MakeHuman™. It is animated for testing purposes, some poses of the animation are shown.

Figure 8.2: The human model used for testing, in rest pose (left) and in two poses

8.3 Desparsification

Certain pictures of the the mesh showing how many bones influence each vertex: before (Figure 8.3) and after (Figure 8.4) the desparsification algorithm. Red, a vertex is influenced by one bone; orange, by two bones; yellow, by three; green, by four bones (the desired amount); blue, by more than five bones.

Moreover, we now are going to show further images, with a single bone: the vertices that are influenced by this bone with weights > 0 (green), the vertices that are influenced by it with weights equal to 0 (yellow) and the vertices that are not influenced by it (blue) (Figures 8.5, 8.6, 8.7).

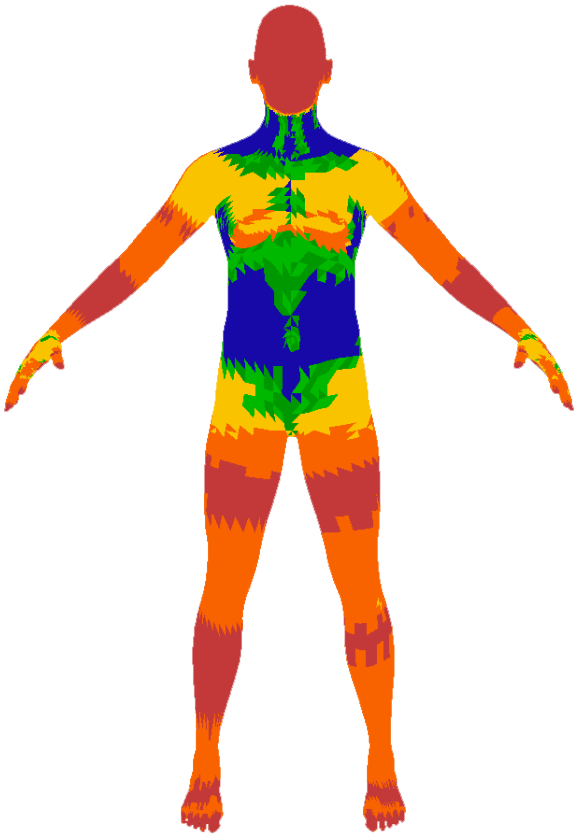

Figure 8.3: Mesh before desparsification

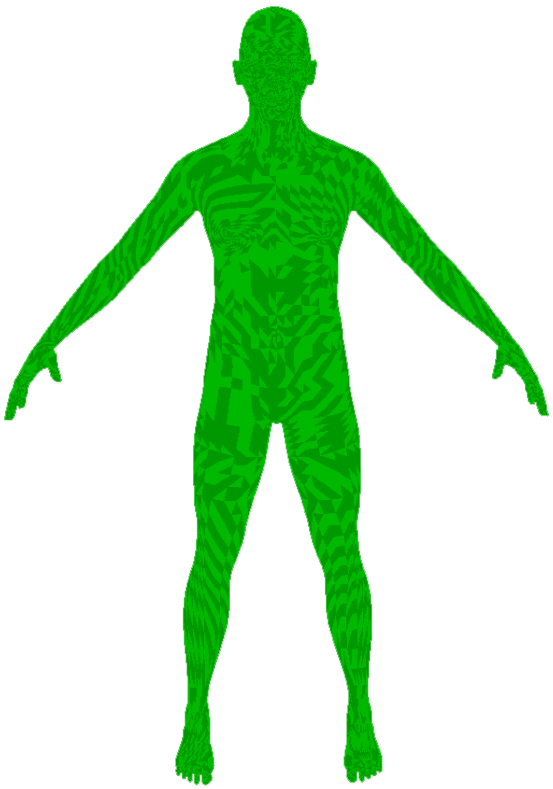

Figure 8.4: Mesh after desparsification

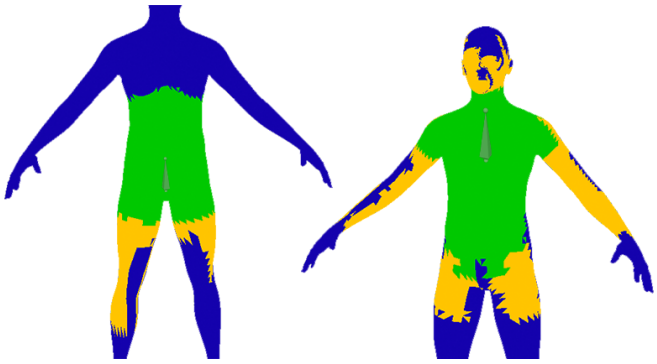

Figure 8.5: Spine and pelvis after desparsification

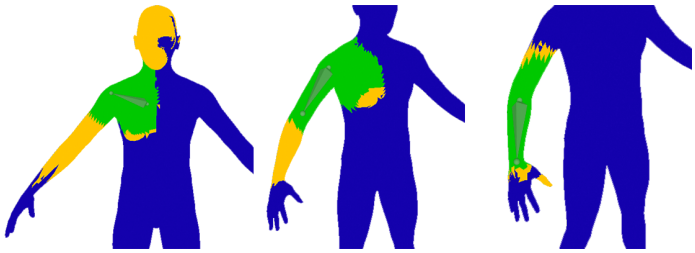

Figure 8.6: Right arm: clavicle, upper arm, lower arm

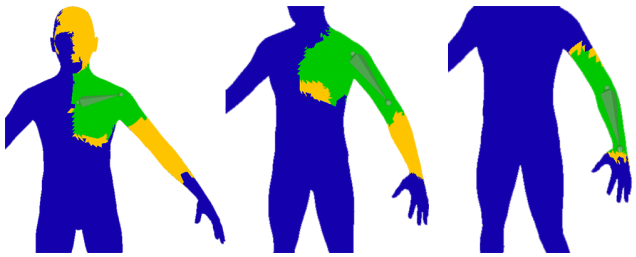

Figure 8.7: Left arm: clavicle, upper arm, lower arm

8.4 Weights computation

Figures 8.8, 8.9, 8.10 and 8.11 represent the four states of a posed mesh during animation sculpting. The user only sees the first and the last state.

Figure 8.12 represents the resulting vertex weights in a different pose than the edited one, in comparison to sculpting without the optimisation.

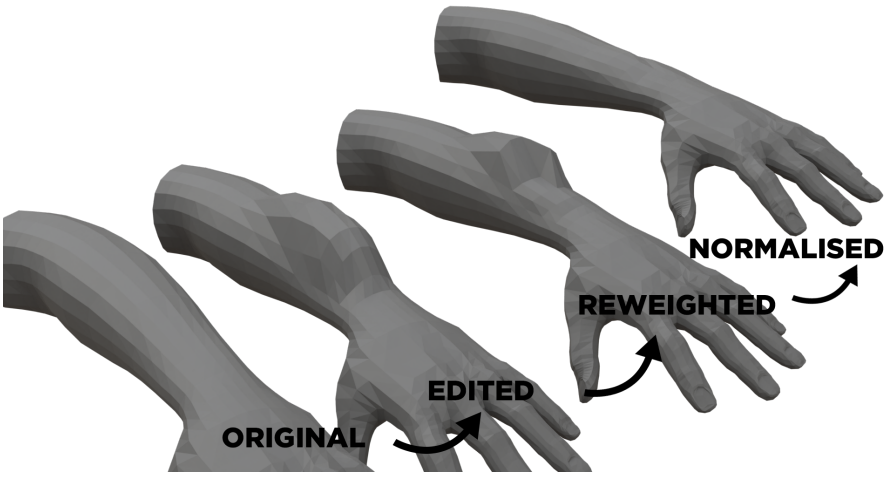

Figure 8.8: Left forearm, first attempt

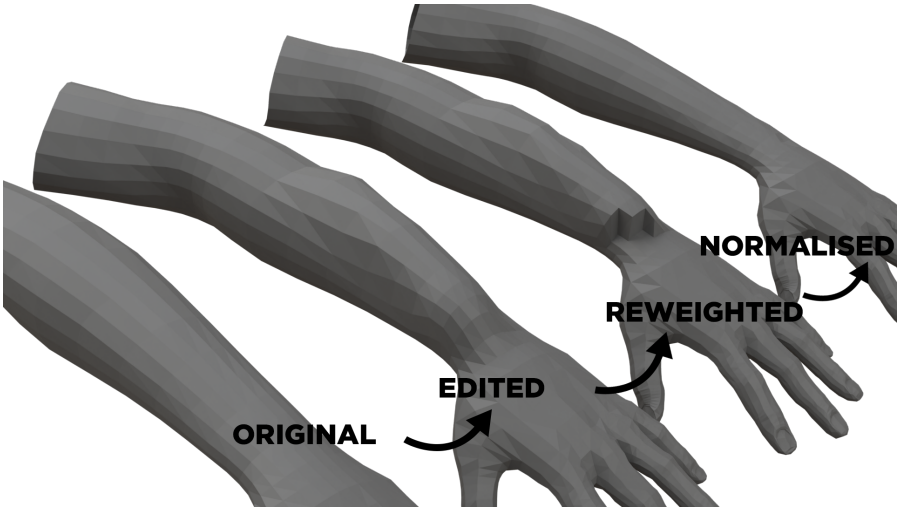

Figure 8.9: Left forearm, second attempt

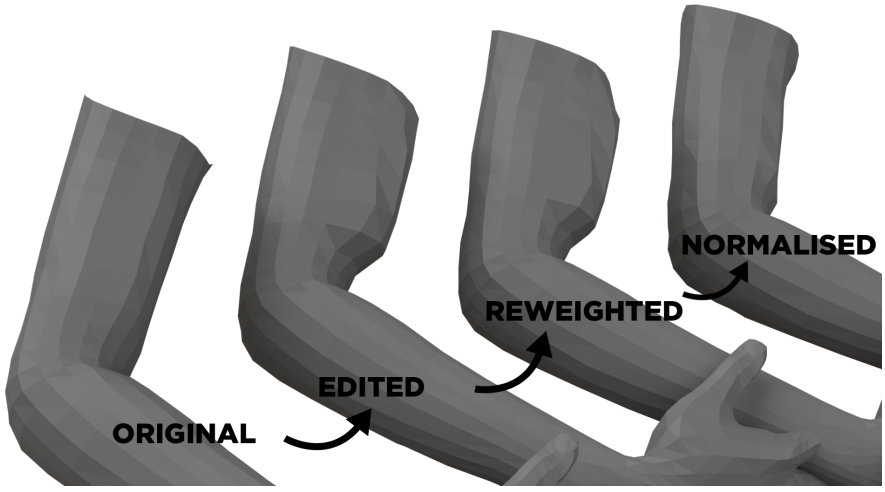

Figure 8.10: Right upper arm

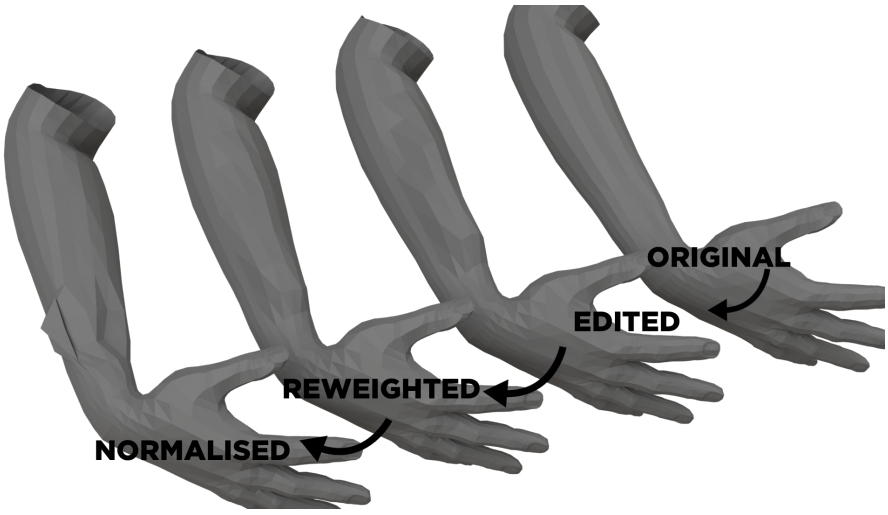

Figure 8.11: Right lower arm

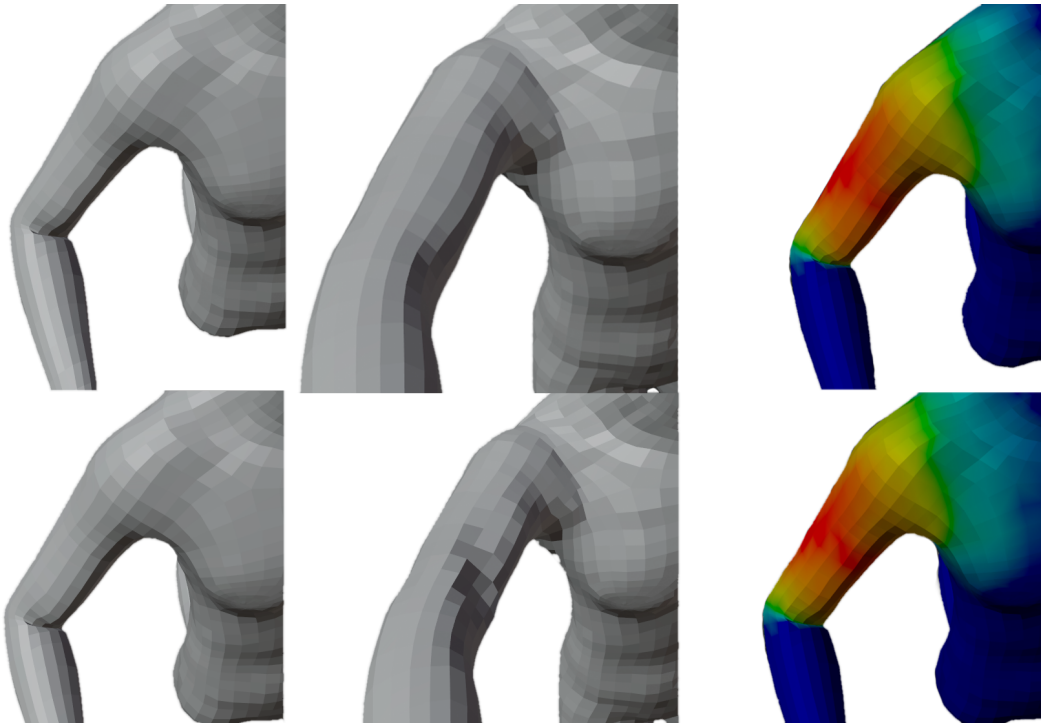

Figure 8.12: The right upper arm before (top row) and after (bottom row) editing (including normalisation). The middle column is the pose that has not been edited.

Chapter 9

Conclusion and future work

Now, we can finally summarise the conclusions of our research, highlighting at the same time certain hints for the future.

We have developed a plug-in: this, clearly, has some limitations. However, this prepares a good framework for any future development. Blender is definitely confirmed to be a platform ready to further experiment on our idea.

The optimising approach works correctly: it computes new weights close to the original editing. However, the re-calculated weights only work in the edited pose, most of the time; the other poses, on the other hand, result in a degenerated shape. This issue may be solved by evaluating weights based upon the ones coming from other poses.

Moreover, Blender only interpolates matrices with the (0;1) bounded range weights. The normalisation process we applied in order to bypass the issue, is not efficient enough. This may be due to a poor choice of new bone slots per vertex inside the desparsification algorithm; or due to our normalisation approach. That being so, it is possible to solve it by changing the desparsification algorithm or by using a solver that already implements said bounds.

Although we established that Blender is ready for the analysis of this project, we can scratch this idea and build an entire new framework that does not require any bound (see idea proposed in section 5.2.1). This would completely eliminate the necessity of normalising. However, it might compromise compatibility with other software. Therefore, said normalisation might be necessary after the optimisation is complete.

In conclusion, there is a limit in modifying just the weights after the optimisation. In fact, there is the possibility to have more degrees of freedom by also modifying the vertices positions in rest pose. This means, constructing a new system with more than three variables. The result of this new inverse problem might be more precise and less prone to errors.

Bibliography

[1] Calabrese, C., Salvati, G., Tarini, M., and Pellacini, F. Csculpt: A system for collaborative sculpting. *ACM Trans. Graph. 35*, 4 (July 2016).

[2] Dunlop, R. *Production Pipeline Fundamentals for Film and Games.* CRC Press, 2014.

[3] Foundation, B. Blender Python API documentation, 2020. Version 2.83; last updated: 06/05/2020.

[4] Got, E. Developement of a new automatic skinning system. Master's thesis, Universitat Politècnica de Catalunya, 2019.

[5] Lewis, J. P., Cordner, M., and Fong, N. Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (2000), pp. 165–172.

[6] Li, J., Lu, G., and Ye, J. Automatic skinning and animation of skeletal models. *The Visual Computer 27*, 6-8 (2011), 585–594.

[7] Ludvigsen, H., and Elster, A. C. Real-time ray tracing using nvidia optix. In *Eurographics (Short Papers)* (2010), pp. 65–68.

[8] MathWorks, T. Fast facts. *Company Overview* (2020).

[9] Perez, F., Granger, B. E., and Hunter, J. D. Python: an ecosystem for scientific computing. *Computing in Science & Engineering 13*, 2 (2010), 13–21.

[10] Spencer, S. *Zbrush digital sculpting human Anatomy.* John Wiley & Sons, 2010.

[11] Tarini, M., Panozzo, D., and Sorkine-Hornung, O. Accurate and efficient lighting for skinned models. In *Computer Graphics Forum* (2014), vol. 33, Wiley Online Library, pp. 421–428.