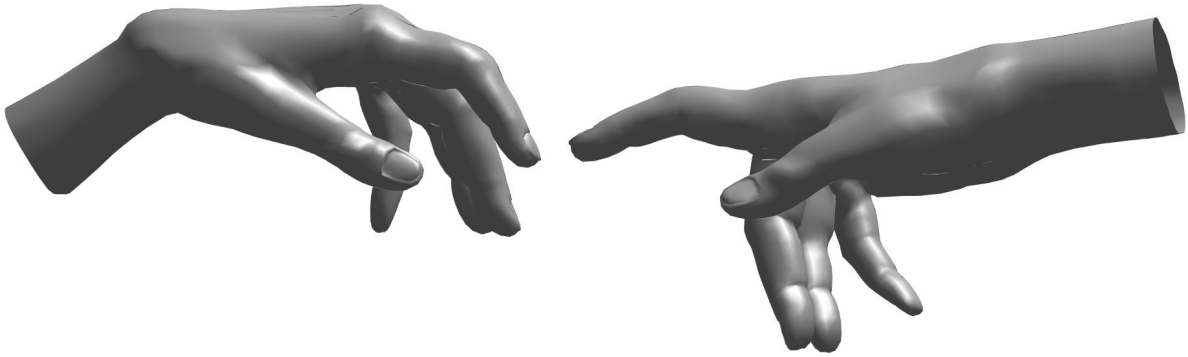
Implicit Skinning

Johan Berdat Laurent Valette Anastasia Tkach Mark Pauly

Computer Graphics and Geometry Laboratory
École Polytechnique Fédérale de Lausanne, Switzerland



Abstract

Mesh deformation is a common problem in 3D rendering and animation. While standard approaches like linear blending or dual quaternion skinning provide high performances, they fail to mimic realistic behaviors. On the other hand, physics-based simulations are not suited for real-time tasks, despite giving high quality results. Rodolphe Vaillant proposes in his thesis a novel way to adjust errors introduced by approximations during the skinning process. Using signed distance functions, his implicit skinning technique fixes and improves bulges. This method is well suited for body animation, hand and fingers in particular. We have studied and implemented his paper in MatLab and C++ to provide real-time animation for hand-tracking.

1 Introduction

Animation heavily relies on mesh manipulation and transformation. As always in engineering, time and quality is a matter of trade-off. Skeletal animation is a popular solution where the mesh surface is attached to a set of interconnected bones. By transforming the skeleton into different poses, the animator is able to deform a mesh in an intuitive way.

Each vertex is associated to one or more bones, according to weights defined by the designer. Traditional skinning techniques combine the rigid transformations of bones to update individual vertices. As such, vertices locations are independent from each others. While this independence allows highly parallelized implementation, the global shape may be affected, resulting in unrealistic configurations. On the other hand, physics-based simulations that are used in offline rendering are computationally too expensive in real-time scenarios. Many solutions were proposed, involving different interpolation strategies and mesh representations. They exhibit some limitations that Vaillant resolves with *implicit skinning* [14].

The first issue, known as the *candy wrapper effect* due to its twisted shrinkage, is induced by bone rotations. Similarly, bone bending also introduces

volume losses. While the former is only present in *linear blend skinning*, *dual quaternion skinning* [12] also fails to solve the latter.

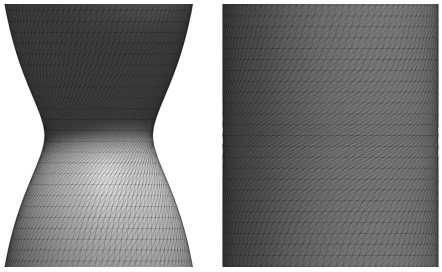


Figure 1: *Illustration of the candy wrapper effect on a twisted cylinder. Dual quaternion and implicit skinning produce the same transformation (right).*

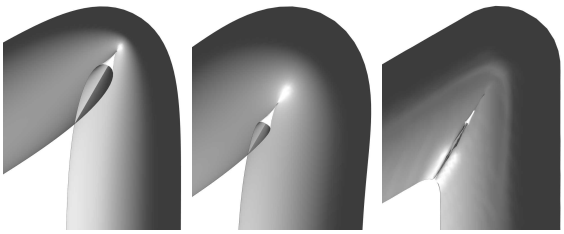


Figure 2: *Illustration of the self-intersection problem on a section of a bended cylinder. Implicit skinning (right) correctly generates a contact region while linear blend skinning (left) and dual quaternion (middle) smoothly fold the mesh. We can also remark the loss of volume that affects linear blending.*

Moreover, these methods are unaware of self-intersections. High deformations, in a realistic scenario, should produce contact regions between skin parts. By nature, these traditional skinning techniques ignore such a behaviour and produce interleaved surfaces.

Another characteristic of implicit skinning is the ability to produce subtle organic effects. In particular, we are interested in generating muscular bulges that appear on the skin when fingers bend. This effect can be seen in figure 3 and generate more plausible deformations around joints.

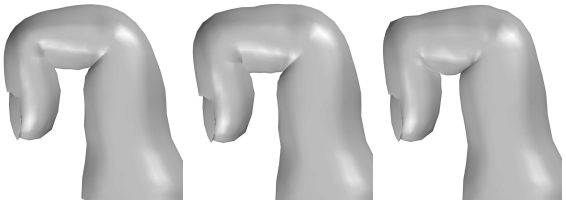


Figure 3: *An index is bent using linear blending (left), dual quaternion (middle) and implicit skinning (right). The mesh is composed of* 1709 *vertices and* 3370 *faces.*

2 Overview

We present a study of the implicit skinning algorithm, by Vaillant et al [14]. The main work is done using MatLab, in order to produce a clean and as efficient as possible implementation. In order to provide a performance measure, a C++ version is also available.

In section 3, we explain in details the theory behind implicit skinning. Actual results and implementations are discussed in section 4.

As our work is meant to be interfaced with a hand-tracking project, some design constraints are present. These are discussed in section 3.1. Once the bone structure is defined, traditional skinning techniques are used to provide an initial guess, in 3.2. This initial guess will be refined and adjusted through implicit skinning. The idea to prevent volume losses is to track and fix surface movements.

Hence, the first step is to estimate the distance to the bones for each vertex, which is explained in sections 3.3 and 3.4. It is done by building a signed distance field around bones and by combining them into a single function. This composition is also the key to bulge generation.

Once these building blocks are defined, the actual algorithm is described in section 3.5. During animation, we apply rigid transformations to the mesh and to the field functions. The initial guess provided by traditional skinning is iteratively corrected. Vertices are moved along field gradients until they reach their original distance. This step, the *surface tracking*, is the core of implicit skinning. It is also during this stage that self-intersections are detected and handled.

Finally, in section 3.6, we discuss an issue related to hand-tracking. The hand model should match

the hand of the user, as poses are highly dependent on bone lengths and volume. Therefore, some methods are discussed to avoid unnecessary computations.

3 Theory

In this section, mathematical details are presented and discussed. On the other hand, implementation subtleties are omitted for readability, but are available in the commented code. It should be noted that some mathematical notations about linear algebra and dual quaternion are defined in the appendices A and B.

3.1 Bone hierarchy

In order to animate a hand, a bone hierarchy must be defined. While following the biological skeleton is an obvious choice for fingers, carpals and metacarpals do not correspond to actual degrees of freedom. Some bending is allowed to bring the thumb and the pinky closer, which must be modeled. As a trade-off, we decided to split the palm in half.

Moreover, this project is associated to hand tracking and must interface smoothly with the computer vision algorithm. More precisely a model-based registration algorithm is used to track user movements. The hand pose is tracked using spheres as control points, which serve as inputs for our animation rendering. Hence, our bone design choice is closely related to the spheres placement. Since the skinning rig only allows rigid deformations, this may also guarantee realistic motion even if some spheres are inaccurately tracked.

In any case, this choice does not affect the remaining pipeline. As long as bone centers and vertices skinning weights are properly defined, the implicit skinning method does not depend on any particular topology. While several techniques exist to automatically compute weights, this critical step requires manual adjustments. Automatic weighting algorithms indeed provide a good guess, but some parts of the palm and thumb ends up with unrealistic weights.

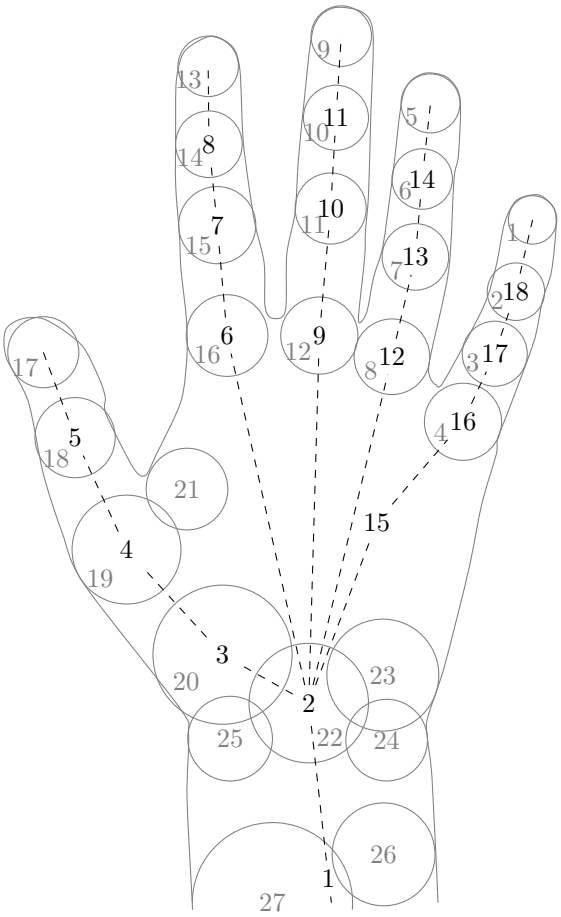


Figure 4: *Schema of control spheres (in gray) and bone hierarchy (in black).*

3.2 Traditional skinning

By traditional, we refer to common real-time techniques used in animation. More precisely, *linear blend skinning* and *dual quaternion skinning* [12] were applied to provide the initial guess required by implicit skinning. Both methods use the skinning weights $w_{b,i}$ associated to each bone b and each vertex i. We compute the transformation $\mathbf{T}_i$ of a vertex as a function of $\mathbf{T}_b$ and $\mathbf{d}_b$, which are respectively the matrix and dual quaternion representing the transformation of the bone b.

$$\mathbf{T}_i = \sum_b w_{b,i}\mathbf{T}_b \qquad \mathbf{d}_i = \overline{\sum_b w_{b,i}\mathbf{d}_b}$$

While $\mathbf{T}_i$ is a 4×4 rigid transformation matrix, $\mathbf{d}_i$ is a *dual quaternion*, which are discussed in appendix B. In both case, vertices $\mathbf{v}_i$ and their normals $\mathbf{n}_i$ can be computed independently:

$$\begin{bmatrix}\mathbf{v}'_i\\1\end{bmatrix} = \mathbf{T}_i \begin{bmatrix}\mathbf{v}_i\\1\end{bmatrix} \qquad \begin{bmatrix}\mathbf{n}'_i\\0\end{bmatrix} = \left(\mathbf{T}_i^{-1}\right)^{\intercal} \begin{bmatrix}\mathbf{n}_i\\0\end{bmatrix}$$

$$\hat{\mathbf{v}}'_i = \mathbf{d}_i \hat{\mathbf{v}}_i \mathbf{d}_i^* \qquad \hat{\mathbf{n}}'_i = \mathbf{d}_{ri} \hat{\mathbf{n}}_i \mathbf{d}_{ri}^*$$

3.3 Mesh reconstruction

The main idea behind implicit skinning is to adjust positions computed using traditional skinning methods. More precisely, each vertex is moved to reach its original distance to the bones. Hence, we need an efficient way to compute a distance for each point in space with respect to each bone.

A *global distance function* $d(\mathbf{p})$ for the surface $\mathcal{S}$ is a *signed distance function* such that $d(\mathbf{p}) = 0$ if and only if $\mathbf{p} \in \mathcal{S}$. Moreover, $d(\mathbf{p}) < 0$ holds for any point inside the surface. Using such representation, any point $\mathbf{p}$ can be evaluated and moved along the gradient $\frac{\partial}{\partial \mathbf{p}} d$ until it reaches the desired distance.

3.3.1 Hermite Radial Basis Function

Radial Basis Function is a surface reconstruction technique widely used in computer graphics. As Vaillant proposes in *Implicit Skinning*[14], we use *Hermite RBF* [13] to compute the signed distance function.

Assuming m centers $\mathbf{c}_k$ and their coefficients α_i and $\boldsymbol{\beta}_k$, the HRBF distance at $\mathbf{p}$ is defined as:

$$d(\mathbf{p}) = \sum_{k=1}^{m} \left(\alpha_k \phi\left(\|\boldsymbol{\delta}_k\|\right) + \boldsymbol{\beta}_k^T \phi'\left(\|\boldsymbol{\delta}_k\|\right) \bar{\boldsymbol{\delta}}_k \right)$$

where $\boldsymbol{\delta}_k = \mathbf{p} - \mathbf{c}_k$. The notation $\bar{\mathbf{v}}$ is used to represent the normalized vector $\mathbf{v}$. Some notations and general relations are described in appendix A. The gradient, which will be used during vertex projection, is therefore computed as:

$$\frac{\partial}{\partial \mathbf{p}} d(\mathbf{p}) = \sum_{k=1}^{m} \left(\alpha_k \phi'\left(\|\boldsymbol{\delta}_k\|\right) \bar{\boldsymbol{\delta}}_k + \boldsymbol{\beta}_k \frac{\phi'\left(\|\boldsymbol{\delta}_k\|\right)}{\|\boldsymbol{\delta}_k\|} + \boldsymbol{\beta}_k^T \bar{\boldsymbol{\delta}}_k \left(\phi''\left(\|\boldsymbol{\delta}_k\|\right) \bar{\boldsymbol{\delta}}_k - \phi'\left(\|\boldsymbol{\delta}_k\|\right) \frac{\bar{\boldsymbol{\delta}}_k}{\|\boldsymbol{\delta}_k\|} \right) \right)$$

A simple yet efficient kernel ϕ is the *triharmonic* function:

$$\phi(x) = x^3 \qquad \phi'(x) = 3x^2 \qquad \phi''(x) = 6x$$

Hence, the simplified HRBF formulas are:

$$d(\mathbf{p}) = \sum_{k=1}^{m} \left(\alpha_k \|\boldsymbol{\delta}_k\|^3 + 3 \|\boldsymbol{\delta}_k\| \boldsymbol{\beta}^T \boldsymbol{\delta}_k \right)$$

$$\frac{\partial}{\partial \mathbf{p}} d(\mathbf{p}) = 3 \sum_{k=1}^{m} \left(\alpha_k \|\boldsymbol{\delta}_k\| \boldsymbol{\delta}_k + \boldsymbol{\beta}_k^T \bar{\boldsymbol{\delta}}_k \boldsymbol{\delta}_k + \|\boldsymbol{\delta}_k\| \boldsymbol{\beta}_k \right)$$

The coefficients are usually computed with an optimization problem, to fit known points on the surface. Given n points $\mathbf{v}_i$ and their normals $\mathbf{n}_i$, the HRBF should represent them as precisely as possible. Therefore, we should satisfy the following equations, which gives a linear system with $4n$ equations and $4m$ variables:

$$d(\mathbf{v}_i) = 0 \qquad \nabla d(\mathbf{v}_i) = \mathbf{n}_i \qquad \forall i$$

The choice of centers and control points is arbitrary and is the key to proper HRBF reconstruction. As Vaillant suggests, we use Poisson sampling over the surface. In order to automatically handle joints, we use an ad hoc hole filling method based on convex optimization (which is discussed in appendix C). Each hole $\mathcal{H}$ is patched with a regular grid of average valence of 6, which is then inflated. The following convex program is used to tightly fit the patch and minimize the distance between vertices:

$$\begin{aligned} \underset{\mathbf{v}}{\text{minimize}} \quad & \sum_{(i,j)} \|\mathbf{v}_i - \mathbf{v}_j\|_2^2 \\ \text{subject to} \quad & \mathbf{v}_h = \mathbf{v}_h^* \qquad h \in \mathcal{H}, \end{aligned}$$

where $\mathbf{v}_h^*$ are the vertices on the boundary of $\mathcal{H}$.

3.3.2 Spherical primitives

Another approach to signed distance functions relies on approximations by primitives composition. Spheres are a common choice when it comes to simple geometrical objects. Intuitively, the distance and gradient of a sphere of center $\mathbf{c}$ and radius r are given by:

$$d(\mathbf{p}) = \|\mathbf{p} - \mathbf{c}\| - r$$

$$\frac{\partial}{\partial \mathbf{p}} d(\mathbf{p}) = \overline{\mathbf{p} - \mathbf{c}}$$

To properly approximate fingers, a cylindrical shape is required. We call *bisphere* the convex hull of two spheres. This can be seen as two distinct spheres connected by a truncated cone. To compute the distance and gradient, we need to determine on which part the projection p' lies.

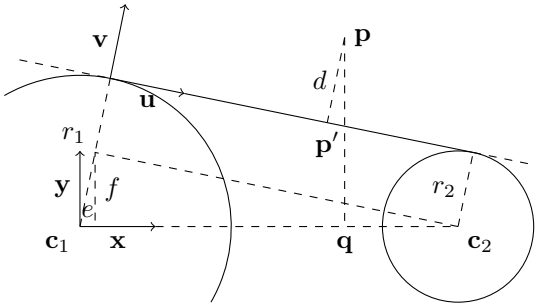


Figure 5: *Schema of a bisphere and relevant metrics used in equations.*

The following derivations are mostly geometrical relations based on figure 5:

$$\mathbf{x} = \overline{\mathbf{c}_2 - \mathbf{c}_1} \qquad \mathbf{y} = \overline{\mathbf{p} - \mathbf{q}}$$

$$\mathbf{q} = (\mathbf{c}_1 - \mathbf{p})^T \mathbf{x}\mathbf{x} \qquad e = \frac{(r_1 - r_2)^2}{\|\mathbf{c}_2 - \mathbf{c}_1\|}$$

$$f = \sqrt{(\|\mathbf{c}_2 - \mathbf{c}_1\| - e)\, e} \qquad \theta = \frac{f}{\|\mathbf{c}_2 - \mathbf{c}_1\| - e}$$

$$\mathbf{u} = \mathbf{x} - \theta \mathbf{y} \qquad \mathbf{v} = \mathbf{y} + \theta \mathbf{x}$$

Using $\mathbf{u}$, we can check whether $\mathbf{p}'$ lies on the conical segment, where the gradient and distance are defined by $\mathbf{v}$. Otherwise, the sphere formulas can be used.

Finally, an additional primitive is needed to represent planar parts, such as the palm. Following the same idea to extend our model, we call *trisphere* the convex hull defined by three spheres. Again, we can use the previous formulas for some cases. If $\mathbf{p}'$ lies on the tangential triangle, then the distance and gradient can be computed using the normal. Otherwise, values are computed using the right bisphere.

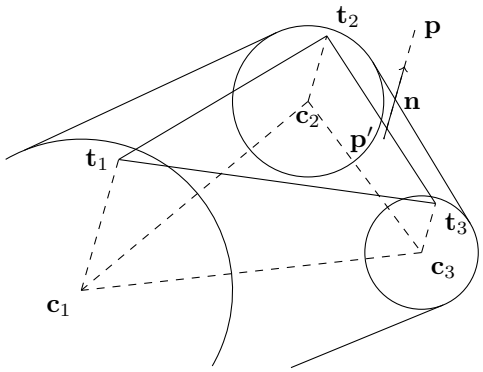


Figure 6: *Schema of a trisphere with the upper tangential plane and its normal.*

3.4 Composition of distance functions

While global distance functions are an intuitive way to represent distances, they have some drawbacks when it comes to combination of multiple surfaces. More precisely, locality is not easily defined. A volume should only influence close neighborhood and should not have any effect after some distance. Hence, we need functions with *local support*, which we call *local distance functions*.

The basic idea is that we define a new function d such that $d(\mathbf{p}) = 1$ inside and $d(\mathbf{p}) = 0$ outside the object, with a smooth transition. The implicit surface is therefore defined by the 0.5-isosurface. As Vaillant suggests, a global distance function can be remapped to provide local support using the following reparametrization, where $l(x)$ is an interpolation function and r is the distance threshold:

$$t_r(d) = \begin{cases} 1 & \text{if } d < -r \\ 0 & \text{if } d > r \\ l(\frac{d}{r}) & \text{otherwise} \end{cases}$$

$$l(x) = -\frac{3}{16}x^5 + \frac{5}{8}x^3 - \frac{15}{16}x + \frac{1}{2}$$

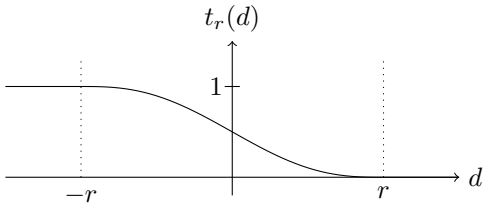


Figure 7: *Plot of the reparametrization function.*

Using global distance functions, each bone can be reconstructed and reparametrized. To compute the distance to the whole mesh, a single distance function must be defined. To combine bone fields, we apply a two steps composition. First, for each pair of neighboring bones (i, j), we combine the two fields into one $f_{ij} = g_{ij}(f_i, f_j)$ using an *operator*. Then, the fields f_{ij} are combined into a single function f using the *union operator*.

3.4.1 Union operator

An important and yet simple operator is the *union*. As the name implies, this composition function aims to merge two surfaces together. Due to the nature of local distance functions, this is achieved in a straightforward manner:

$$g_u(f_1, f_2) = \max(f_1, f_2)$$

As stated before, the gradient of distance functions is required during surface tracking. While Gourmel et al. [8] thoroughly discuss continuity issues by introducing quasi-C^∞ operators to ensure mathematical correctness, we are not going to provide a continuous approximation of the union operator. In practice, the maximum is implemented by using the largest distance, since the singularity is numerically never reached.

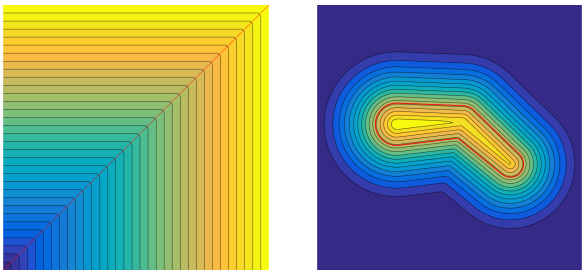


Figure 8: *Illustration of the union operator on the left, where the axes are f_1 and f_2. The red diagonal shows the discontinuity between gradients, which produces the sharp union on the right. The graph on the left represents how g_u depends on f_1 and f_2, which are respectively abscissa and ordinate. Vertical lines are isosurfaces of f_1 and horizontal lines are those of f_2. The limits of both axis are $[0, 1]$ since the field functions are compactly supported.*

3.4.2 Blend operator

A major aspect of Vaillant's work is about producing realistic skin behavior. As the finger bends, the tissue on the inner part of the joint is compressed and produces a bulge. Hence, another composition function is required, the *blend operator*.

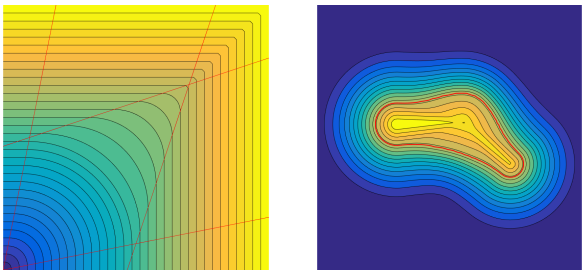


Figure 9: *Illustration of the blend operator on the left. The trapezoidal region produces higher values than the union operator, giving a bulging effect on the right.*

While the behavior is similar to the union for distant configurations, the distance field is increased when both fields are close. The red lines, which are given by the following equations, delimit the boundary between union and bulging. They are controlled by a parameter θ, which is the angle of boundary lines.

$$f_2 = \tan(\theta) f_1 \qquad f_1 = \tan(\theta) f_2$$

$$f_2 = \frac{1}{2}(7 - 5\tan(\theta))f_1 + \frac{7}{4}(\tan(\theta) - 1)$$

$$f_1 = \frac{1}{2}(7 - 5\tan(\theta))f_2 + \frac{7}{4}(\tan(\theta) - 1)$$

The two families of curves in the trapezoidal area are given by solving the following equations with respect to C:

$$(f_1 + a + bC)^2 + (f_2 + a + bC)^2 = (C + a + bC)^2$$

$$a = \begin{cases} 0 & \text{if lower area} \\ -\frac{7}{4}(\tan(\theta) - 1) & \text{if upper area} \end{cases}$$

$$b = \begin{cases} -\tan(\theta) & \text{if lower area} \\ -\frac{1}{2}(7 - 5\tan(\theta)) & \text{if upper area} \end{cases}$$

To determine in which area the pair (f_i, f_j) lies, we use the following tests, which are true in the upper area:

$$\frac{\left(f_1 - \frac{\tan(\theta)}{2}\right)^2 + \left(f_2 - \frac{\tan(\theta)}{2}\right)^2}{\left(\tan(\theta) - \frac{\tan(\theta)}{2}\right)^2} > 1$$

$$f_1 > \frac{\tan(\theta)}{2} \text{ or } f_2 > \frac{\tan(\theta)}{2}$$

The parameter θ is used to control the blending strength. It it usually computed according to the angle α, using the following mapping:

$$\theta(\alpha) = \begin{cases} \theta_0 & \text{if } \alpha \leq \alpha_0 \\ l_0(\alpha) & \text{if } \alpha \in]\alpha_0, \alpha_1] \\ l_1(\alpha) & \text{if } \alpha \in]\alpha_1, \alpha_2] \\ \theta_2 & \text{if } \alpha > \alpha_2 \end{cases}$$

$$l_0(\alpha) = \left(\kappa\left(\frac{\alpha - \alpha_1}{\alpha_0 - \alpha_1}\right)\right)^{\omega_0}(\theta_0 - \theta_1) + \theta_1$$

$$l_1(\alpha) = \left(\kappa\left(\frac{\alpha - \alpha_1}{\alpha_2 - \alpha_1}\right)\right)^{\omega_1}(\theta_2 - \theta_1) + \theta_1$$

$$\kappa(x) = 1 - \exp\left(1 - \frac{1}{1 - \exp(1 - \frac{1}{x})}\right)$$

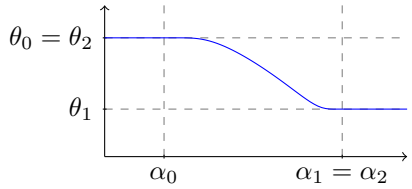


Figure 10: *Plot of θ with respect to α, with $\alpha_0 = \frac{\pi}{8}$, $\alpha_1 = \alpha_2 = \frac{\pi}{2}$, $\theta_0 = \frac{\pi}{4}$, $\theta_1 = \theta_2 = \frac{\pi}{10}$, $\omega_0 = \frac{1}{2}$, $\omega_1 = 1$, which produces realistic finger bulges. In practice, these settings mean that we use the union for angles up to α_0. Then, the bulge increases up to a maximum and it is kept for higher angles.*

The angle α can be defined as the angle between bones. However, this naive approach does not distinguish the outer and inner sides of the joint, as shown in figure 9. Some strategy must therefore be used to disambiguate the two cases. Vaillant quickly describes α as the angle between the gradients of f_1 and f_2. This blend strategy assumes that gradients are collinear on the outer side of joints, while separated by an angle close to α on the inner side. Unfortunately, as we can see in figure 11, this is not true in most scenarios, which can lead to some artifacts. To solve this issue, we define in the rest pose a *blending weight* between 0 and 1 for each vertex. The inner part of fingers have higher values to produce bulges, while most of the hand is zero to have pure union. The computation of this coefficient is based on local bone coordinates of vertices.

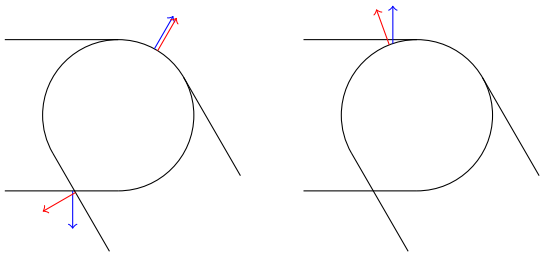


Figure 11: *Schema of distance function gradients. On the left, gradients are collinear on the outer part of the joint. On the right, this is no longer true, because the blue vector is produced by the projection on a cylinder.*

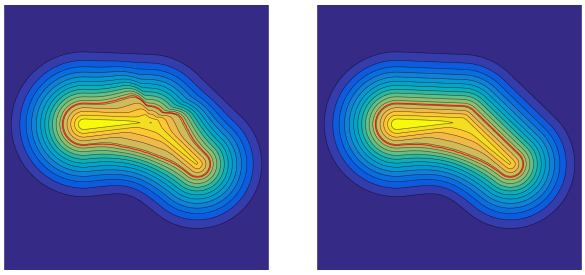


Figure 12: *Illustration of the blend strategies. On the left, the angle between gradients is used to blend the whole finger. On the right, the angle between bones is used to blend the lower part only.*

The blend operator gradient could be analytically computed, but the performance cost is too high. In practice, to reach high performances, gradients and signed distance function are discretized and precomputed.

3.5 Surface tracking

Surface tracking is used during animation to correct the mesh deformation produced either by linear blend or dual quaternion skinning [12]. The main idea is to project every vertex such that its isovalue is the same as it was in the rest pose.

In order to do this, two steps are repeated until convergence: *vertex projection* and *tangential relaxation*. Finally, *laplacian smoothing* is applied on contact regions to reduce sharp features.

3.5.1 Vertex projection

During the vertex projection step, each vertex $\mathbf{v}_i$ is moved along the gradient of the field function f, until it reaches its original isovalue iso_i. We track the isovalues in the rest pose instead of the 0.5-isosurface to avoid losing the details of the mesh. Newton iterations are used, with $\sigma = 0.35$, to provide a robust, fast and accurate convergence:

$$\mathbf{v'}_i = \mathbf{v}_i + \sigma(iso_i - f(\mathbf{v}_i))\frac{\nabla f(\mathbf{v}_i)}{\|\nabla f(\mathbf{v}_i)\|}$$

The main convergence criterion is when the vertex has reached its isovalue at rest:

$$|iso_i - f(\mathbf{v}_i)| < \varepsilon$$

Additionally, in order to avoid generating self-intersections, we should stop iterating when vertices reach a contact region. To detect this case, Vaillant proposes the following test.

For every vertex, we know the gradient at the current iteration and at the previous one. If the angle between these two vectors is greater than 55°, we consider that the vertex lies on a contact surface. The idea is that the gradient will flip when a vertex crosses from one implicit surface to another.

3.5.2 Tangential relaxation

This step is used to improve the distribution of the vertices. High distortions can be introduced by the vertex projection. Vertices are therefore moved toward the weighted centroid of their neighbours. Barycentric coordinates Φ_{ij} are precomputed in the rest pose for each vertex $\mathbf{v}_i$ with respect to its one-ring neighbours. After each vertex projection, the tangential relaxation relocates vertices to avoid drift:

$$\mathbf{v}'_i = (1-\mu)\mathbf{v}_i + \mu\sum_j \Phi_{ij}\mathbf{q}_{ij},$$

where $\mathbf{q}_{ij}$ are the one-ring neighbours of $\mathbf{v}_i$ projected onto its the tangential plane and the coefficient μ is used to control the strength of the operation. It is defined as a function of the distance to the isovalue in the rest pose, so that it decreases as the vertex approaches convergence:

$$\mu = \max\left(0, 1 - (|iso_i - f(\mathbf{v}_i)| - 1)^4\right)$$

Since computing generalized barycentric coordinates is not trivial, we give a summary of the method we used for this: the *mean value coordinates* [9]. Given a vertex $\mathbf{v}$, the goal is to find weights Φ_j such that $\mathbf{v} = \sum_j \Phi_j \mathbf{q}_j$, with $\mathbf{q}_j$ the one-ring neighbours projected onto the tangential plane of $\mathbf{v}$.

First, let's assume we can traverse the neighbours cyclically. We compute the weight of each neighbour j as:

$$\Phi_j = \frac{\tan(\alpha_{j-1}/2) + \tan(\alpha_j/2)}{r_j}$$

The weights are then normalized such that $\sum_j \Phi_j = 1$. The notations used are described in figure 13.

For the case where the vertex $\mathbf{v}$ is on a mesh boundary, we cannot cycle through the neighbours. To deal with this issue, we do not apply the tangential relaxation on the vertices that are on a mesh boundary.

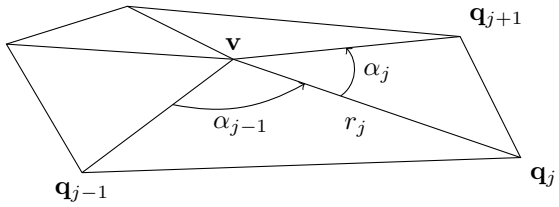


Figure 13: *Notations used for the mean value coordinates.*

3.5.3 Laplacian smoothing

Finally, laplacian smoothing is used to remove sharp features introduced by the self-intersection heuristic. This step must be applied immediately after the last vertex projection:

$$\mathbf{v}_i = (1 - \beta_i)\mathbf{v}_i + \beta_i \tilde{\mathbf{v}}_i$$

Each vertex $\mathbf{v}_i$ is moved toward the centroid $\tilde{\mathbf{v}}_i$ of its one-ring neighborhood. The smoothing coefficient β_i is set to 1 for vertices stopped at a gradient discontinuity, 0 for the others. To have a clean transition, the values β_i are interpolated using diffusion prior to applying the smoothing.

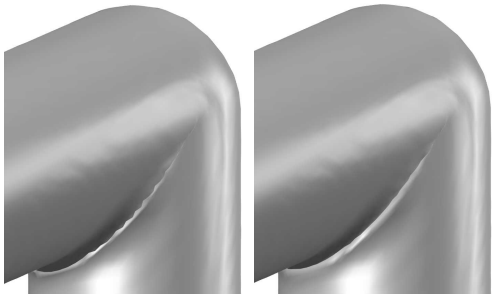


Figure 14: *Without smoothing (left), the mesh surface exhibits a sawtooth pattern. Laplacian smoothing removes these oscillations.*

3.6 Hand registration

Since this project is associated to hand tracking, some issues unrelated to implicit skinning might arise. Sphere positions are used as input of our implementation. These control points are located using computer vision and constrained through model-based registration. Unfortunately, these locations depends on the morphology of the user hand. Incoherent bone lengths is fixed by our pose estimator, in order to match the mesh metrics, which leads to an inaccurate representation. Therefore the hand model should correspond to the user skeleton.

The obvious choice is to regenerate required data for any new user. Many tools provide 3D reconstruction based on pictures, which can be used to generate a point cloud. Usually noisy, this model can then be refined to obtain satisfactory results. More time-consuming techniques can also be applied, like designing the mesh by hand, using modeling tools. But for non-professional purpose, several hours of work by a designer may be inappropriate.

Moreover, some details must be adjusted to fit the new model. The majority of the pipeline does not depends on the actual mesh, as long as the topology is preserved. However, bone locations and vertex weights need to be defined. As this was proven during this project, this is a tedious and non-trivial task.

To solve this issue, we have tried to apply rigid registration, as successfully applied to human faces by Bouaziz[3]. By automatically generating a mapping between our neutral mesh and the user scan, data could be acquired in a cheaper way. Sphere locations and weights could be exported to the new mesh. Furthermore, the clean neutral mesh could be used instead of the noisy and ill-triangulated point cloud.

Figure 15: *A point cloud generated from several pictures. Cleaning this mesh and adjusting bone locations is not a trivial task.*

As the hand has many more degrees of freedom than the face, some changes must be applied to the original algorithm. The main difficulty is to estimate the point cloud pose. We have tried several optimization strategies, without significant success. The most promising approach was highly

specialized feature detection. By first locating the palm, the global size of the hand can be estimated. Then, each phalange is fit sequentially, until the global pose is acquired.

Since this issue was not the top priority of this work, hand registration was too time-consuming to be completed. Good approximations were reached, but without sufficient precision to be a viable alternative to manual design.

Figure 16: *Palm, wrist and some phalanges were fit using the Iterative Closest Point algorithm. Better alternatives could be used, as the Sparse ICP by Bouaziz [4].*

4 Implementation and results

We were given as inputs a mesh of a hand and the positions of the spheres. As explained in the theory, the first part was to define a bone hierarchy and create skinning weights. The placement of the bones was based on the positions of the spheres.

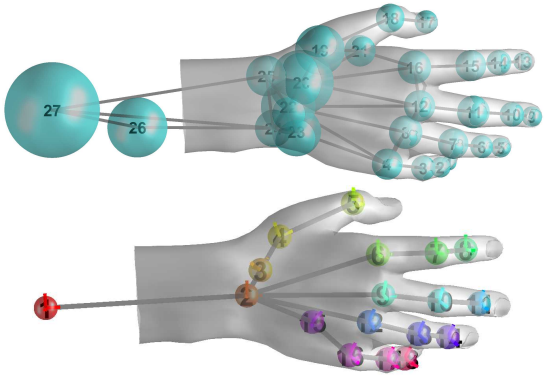


Figure 17: *The hand mesh and the spheres that are provided as inputs (top). The hierarchy of bones (bottom) is based on the positions of the spheres.*

Blender was used to create and tweak the skinning weights. Based on the weights, we were able to partition the mesh around every bone. More precisely, every vertex was assigned to the bone with the highest weight. The following figure illustrates this process.

Figure 18: *The hand mesh and its partitioning. Each color corresponds to the area of influence of a bone.*

This partitioning is needed for the surface reconstruction, which was done with HRBFs and spherical primitives. In the case of the latter, the positions of the spheres are given as inputs and each sphere is assigned to a bone, so there was little work to do to reconstruct the surface.

Getting correct approximations of the mesh surface with HRBFs was a bit more challenging. The first problem we solved was to fill the holes left at the extremities of the partitions. This is important because two neighbouring surfaces should joint without introducing gaps or gradient discontinuities.

Another problem was the choice of the HRBF's centers and control points. Poisson disk sampling on the mesh proved to be a good strategy, because it avoids having multiple samples too tightly packed. Even with this, it may happen that the reconstruction of a part does not smoothly follow the mesh surface. To tackle this problem, we wrote a Matlab script where a reconstruction is showed and the user can accept or reject it, in which case, we resample the mesh. This allows to tweak the surfaces, if needed.

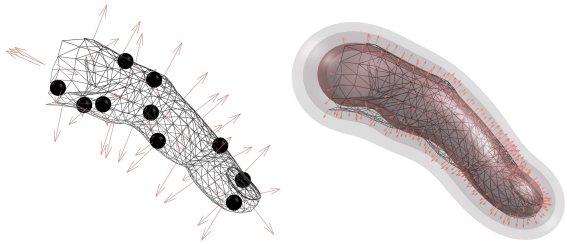


Figure 19: *Illustration on a finger of the HRBF sampling (left) and the surface reconstruction (right). The spheres represent the HRBF centers while the arrows are the positions and normal vectors used as control points.*

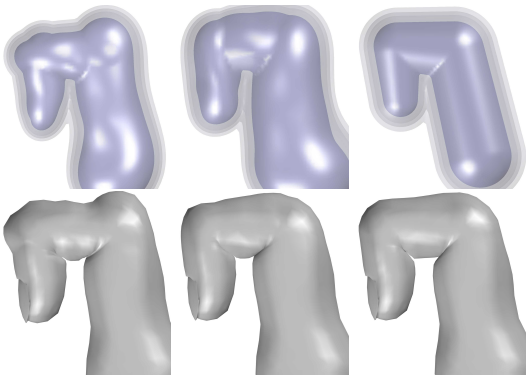


Figure 21: *Close view of an index finger deformed using HRBFs (left and middle) and spherical primitives (right). The top row shows the reconstructed surfaces, combined with the union operator. The leftmost column is showing the results with poorly adjusted HRBFs, while the column in the middle has a smoother surface because we tweaked the HRBFs.*

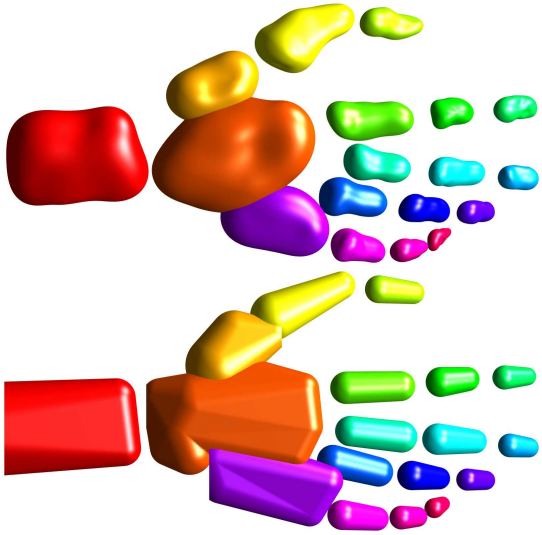


Figure 20: *Reconstruction of the mesh surface using HRBFs (top) and spherical primitives (bottom).*

As we explained, surfaces should slide over each others at joints, in order to avoid gaps. From that regard, the spherical primitives are better, as we can see in the following figure.

However, in the interior side, the mesh is more appealing when using HRBFs, because the bulge effect is more visible. This is caused by the spherical primitives producing a flat surface in-between the spheres, thus projecting all the vertices in the same direction.

In some poses, the spherical primitives can also suffer from the problem of surfaces non smoothly joining. Indeed, the wrist, the palm and the bases of the thumb and the little finger are composed of flat surfaces. So it can happen that these surfaces are misaligned. This is demonstrated in the following figure, where we can see a bump near the wrist.

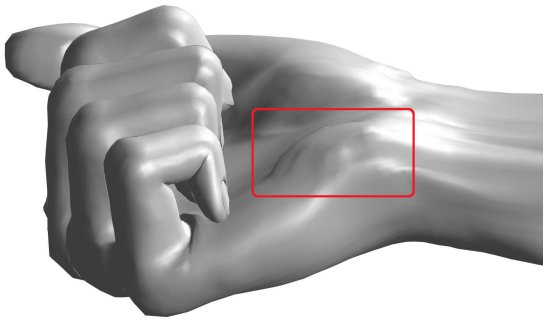


Figure 22: *A hand animated with spherical primitives.*

During animation, the choice of the geometric skinning method used to get an initial deformation

is one of the parameters that can be used to influence the result. We support linear blending and dual quaternion, but other techniques could easily be plugged in. By default, we use dual quaternion skinning because it does not suffer from the candy wrapper effect and it is less likely to produce deep self-intersections. As illustrated in the following figure, self-intersections could lead to projections in the wrong direction.

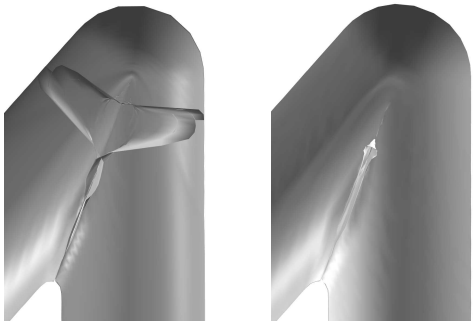


Figure 23: *When the initial mesh deformation is produced by linear blend skinning (left), this section of a cylinder exhibits artifacts because some vertices were projected on the wrong side. With dual quaternion (right), the vertex projection works correctly because the vertices are on the correct side of the implicit surfaces.*

Another parameter is the reparametrization distance used to convert the fields with global supports into locally supported fields. Its effect is to increase or decrease the bulge, as we can see in the following figure where thresholds of 1, 2 and 3 were used.

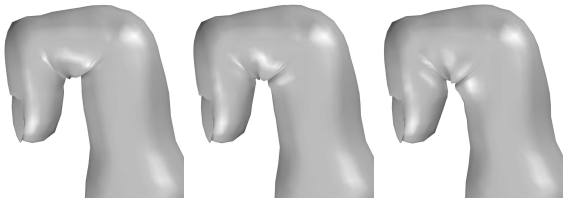


Figure 24: *Index finger deformed while varying the threshold. We can see that the muscular bulges inflate as the threshold increases.*

Fine control over the bulge effect is provided by the 8 arguments of the function $\theta(\alpha)$. They can be changed to model different types of material, like rubber or skin. Because it is quite tedious to set these correctly and because we were interested only in hand deformations, we generally used the parameters shown in figure 10.

Finally, in order to avoid creating bulges on the exterior side of fingers, we added a parameter the we call *blending weights*. They are used to linearly interpolate between union and blend. As illustrated in the following figure, we set the weights close to 1 in the inner part of the fingers while the exterior side is set to 0. The advantage is that the use of the weights creates a really smooth field, allowing for realistic rendering. On the other hand, an unexpected side effect of the interpolation is that it drastically reduces the influence of other parameters.

Figure 25: *Illustration of the blending weights on the hand. The intensity of orange indicates the strength with which we apply the blend operator.*

We implemented implicit skinning both in MatLab and C++. In MatLab, all the functions are vectorized over the vertices, allowing for decent performances (not real-time though). We compared the versions using a low polygon mesh and with a tessellated mesh. The low-resolution mesh has 3 045 vertices while the high-resolution has 12 125 vertices. It should be noted that the C++ version only supports the union operator and HRBFs, so we tested the MatLab version both with these settings and with the bulge operator for the fingers.

	HRBF	Spheres	DQS
ML low-res	3.0s	5.6s	15ms
ML high-res	5.9s	10.6s	90ms
ML low (union)	0.75s	1.4s	-
ML high (union)	1.45s	2.15s	-
C++ low-res	13ms	-	1.5ms
C++ high-res	55ms	-	1.7ms

Table 1: *Comparison of the time to compute one animation in MatLab and in C++.*

5 Conclusion

As we can see in our results, implicit skinning has significative visual improvements over dual quaternion skinning. Even if no bulge effect is applied, using only the union operator fixes vertices distances to bones. Soft bodies that rely on a rigid skeleton benefit from this algorithm and display more realistic behaviour. The blend operator and the versatile nature of isosurfaces combination makes Implicit skinning a promising asset in computer graphics.

For clarity's sake, both our codes use straightforward implementations. Hence, many optimization opportunities are left. Profiling our code revealed that the most time-consuming task is signed distance function evaluation. Precomputing and caching these values would probably dramatically improves performances. However, code readability is a major goal of our project.

As Vaillant suggests, GPU implementation is required to achieve maximal framerate. Due to the iterative nature of the method, parallelization is not as trivial as in linear blend or dual quaternion skinning. Each tangential relaxation step requires the locations of neighboring vertices. This memory barrier might be the bottleneck in a distributed implementation.

Our implementation also suffers from some minor issues. As previously discussed, many parameters must be finely tuned in order to provide realistic results. Skinning weights, HRBF coefficients and reparametrization threshold can still be improved. Moreover, some heuristics could be enhanced, such as bone transforms estimation from spheres. The thumb orientation in highly curved poses might be inaccurately reproduced.

In conclusion, implicit skinning is a powerful tool that requires more care than usual skinning techniques. As it enforces rigid transformation more tightly, it is especially suited for hand animation. With appropriate optimizations, this algorithm provides real-time animations that requires realistic results.

Acknowledgments

This work was done as a semester project, for our Master in Computer Science at the École Polytechnique Fédérale de Lausanne, Switzerland. We thank Anastasia Tkach and Mark Pauly for their support during the whole semester. We also thank Alec Jacobson for his amazing MatLab toolbox [11].

A Matrix calculus

To avoid tedious details during explanations, mathematical expressions of basic building blocks will be described here. Most of the derivations can be formulated using matrices and are based on both analysis and geometrical results. While common calculus concepts work as expected, such as derivative chain rule, most complex cases need some care in order to produce clean expressions.

$$\|\mathbf{v}\| = \sqrt{\mathbf{v}^T\mathbf{v}} \qquad \frac{\partial}{\partial \mathbf{v}} \|\mathbf{v}\| = \frac{\mathbf{v}^T}{\|\mathbf{v}\|} = \bar{\mathbf{v}}^T$$

$$\frac{\partial}{\partial \mathbf{v}} \bar{\mathbf{v}} = \frac{1}{\|\mathbf{v}\|} \left(\mathbf{I}_3 - \frac{\mathbf{v}}{\|\mathbf{v}\|} \frac{\mathbf{v}^T}{\|\mathbf{v}\|} \right) = \frac{\mathbf{I}_3 - \bar{\mathbf{v}}\bar{\mathbf{v}}^T}{\|\mathbf{v}\|}$$

$$\mathbf{v} \times \mathbf{w} = \begin{bmatrix} v_y w_z - v_z w_y \\ v_z w_x - v_x w_z \\ v_x w_y - v_y w_x \end{bmatrix} = [\mathbf{w}]_\times^T \mathbf{v}$$

$$\frac{\partial}{\partial \mathbf{v}} \mathbf{v} \times \mathbf{w} = \begin{bmatrix} 0 & w_z & -w_y \\ w_z & 0 & w_x \\ -w_y & w_x & 0 \end{bmatrix} = [\mathbf{w}]_\times$$

B Dual quaternion

A *quaternion* is an extension to complex numbers usually used to represent 3D rotations. While we assume basic knowledge about their definition and usage, expressions required for skinning are described in this appendix. We adopt the vector notation by packing imaginary components into a vector:

$$\mathbf{q} = \begin{bmatrix} q_s \\ \mathbf{q}_v \end{bmatrix} \qquad \mathbf{q} + \mathbf{r} = \begin{bmatrix} q_s + r_s \\ \mathbf{q}_v + \mathbf{r}_v \end{bmatrix}$$

$$\mathbf{qr} = \begin{bmatrix} q_s r_s - \mathbf{q}_v^T \mathbf{r}_v \\ q_s \mathbf{r}_v + r_s \mathbf{q}_v + \mathbf{q}_v \times \mathbf{r}_v \end{bmatrix}$$

The conjugate is used to define the norm:

$$\mathbf{q}^* = \begin{bmatrix} q_s \\ -\mathbf{q}_v \end{bmatrix} \qquad \|\mathbf{q}\| = \sqrt{\mathbf{q}\mathbf{q}^*} \qquad \bar{\mathbf{q}} = \frac{\mathbf{q}}{\|\mathbf{q}\|}$$

Unit quaternions have a length of 1 and are used to represent any rotation. A vector $\mathbf{v}$ is transformed by a unit quaternion $\mathbf{q}$ as follows:

$$\begin{bmatrix} 0 \\ \mathbf{v}' \end{bmatrix} = \mathbf{q} \begin{bmatrix} 0 \\ \mathbf{v} \end{bmatrix} \mathbf{q}^*$$

To represent translations as well, *dual quaternions* can be used, which are a pair of quaternions. Again, details about *dual numbers* will not be covered in this document. As a reminder, the *dual operator* ε is defined such that $\varepsilon^2 = 0$, but $\varepsilon \neq 0$. Quaternion arithmetics is unchanged:

$$\mathbf{d} = \mathbf{d}_r + \mathbf{d}_d\varepsilon \qquad \mathbf{d} + \mathbf{e} = \mathbf{d}_r + \mathbf{e}_r + (\mathbf{d}_d + \mathbf{e}_d)\,\varepsilon$$

$$\mathbf{de} = \mathbf{d}_r\mathbf{e}_r + (\mathbf{d}_r\mathbf{e}_d + \mathbf{d}_d\mathbf{e}_r)\,\varepsilon$$

$$\mathbf{d}^* = \mathbf{d}_r^* + \mathbf{d}_d^*\varepsilon \qquad \|\mathbf{d}\| = \sqrt{\mathbf{d}\mathbf{d}^*} \qquad \bar{\mathbf{d}} = \frac{\mathbf{d}}{\|\mathbf{d}_r\|}$$

A point $\mathbf{v}$ can be represented by the following dual quaternion:

$$\hat{\mathbf{v}} = \begin{bmatrix} 1 \\ \mathbf{0} \end{bmatrix} + \begin{bmatrix} 0 \\ \mathbf{v} \end{bmatrix} \varepsilon$$

The dual quaternion $\mathbf{d}$ associated to rotation $\mathbf{q}$ and translation $\mathbf{t}$ is given by:

$$\mathbf{d} = \mathbf{q} + \frac{1}{2}\mathbf{q}\begin{bmatrix} 0 \\ \mathbf{t} \end{bmatrix}\varepsilon \qquad \hat{\mathbf{v}}' = \begin{bmatrix} 1 \\ \mathbf{0} \end{bmatrix} + \begin{bmatrix} 0 \\ \mathbf{v}' \end{bmatrix}\varepsilon = \mathbf{d}\hat{\mathbf{v}}\mathbf{d}^*$$

C Quadratic programming

In convex optimization, a *quadratic program* with *linear equality constraints* can be represented as:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \frac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{c}^T\mathbf{x} \\ \text{subject to} \quad & \mathbf{E}\mathbf{x} = \mathbf{d} \end{aligned}$$

The solution can be found by solving the *equilibrium equations*:

$$\begin{bmatrix} \mathbf{Q} & \mathbf{E}^T \\ \mathbf{E} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} -\mathbf{c} \\ \mathbf{d} \end{bmatrix}$$

Since we usually deal with energy minimization, the following least square formulation is more appropriated, where the first h variables are constrained

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \frac{1}{2}\sum_c w_c \left\|\mathbf{A}_c\mathbf{x}\right\|_2^2 \\ \text{subject to} \quad & x_i = x_i^* \qquad i = 1, ..., h \end{aligned}$$

This program can be solved by using:

$$\mathbf{Q} = \sum_c w_c \mathbf{A}_c^T\mathbf{A}_c$$

$$\mathbf{c} = 0 \qquad \mathbf{E} = \begin{bmatrix} \mathbf{I}_h & \mathbf{0} \end{bmatrix} \qquad \mathbf{d} = \mathbf{x}^*$$

D MatLab code

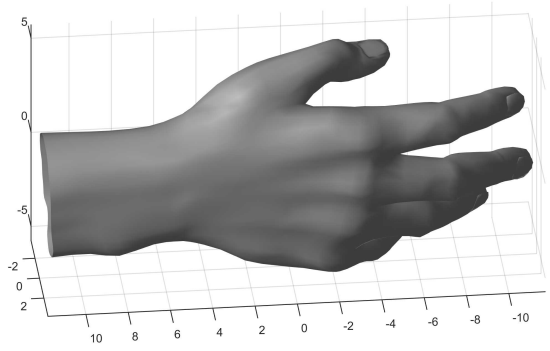
In this appendix, the MatLab implementation will be discussed and explained. Some examples will be presented to provide an overview of the work. Implementation details are available as comments in the source code.

As most of the project relies on geometry processing, the code is designed to handle large amount of data. MatLab efficiency heavily depends on code vectorization, i.e. applying operations on matrices rather than on individual values. Therefore, we assume that the reader has some knowledge of MatLab scripting and indexing subtleties.

First, the *Geometry Processing Toolbox*[11] by Alec Jacobson must be introduced. Developed during its PhD, this project has benefited from various collaborators. Many mesh algorithms are available, combining efficiency and ease-of-use, which makes this library a strong ally in any computer graphics project.

We adopted most of our convention according to *gptoolbox*. *Vectors* of size n are stored as $1 \times n$ matrices, which is the transpose of the mathematical representation. *Lists* of size n are collections of scalars or vectors and are stored as $n \times d$ matrices. *Meshes* in their minimal representation are stored as a list of vertex positions `V` and a list of face indices `F`.

```
[V, F] = load_mesh('neutral.obj');
```



D.1 Hermite Radial Basis Function

Poisson sampling over the surface to generate centers and control points is respectively done with `hrbf_centers` and `hrbf_points`. The system of equations described in section 3.3.1 is solved by `hrbf_coefficients`. The following code generate 20 centers `C` and several control points `P` and compute HRBF coefficients `K`:

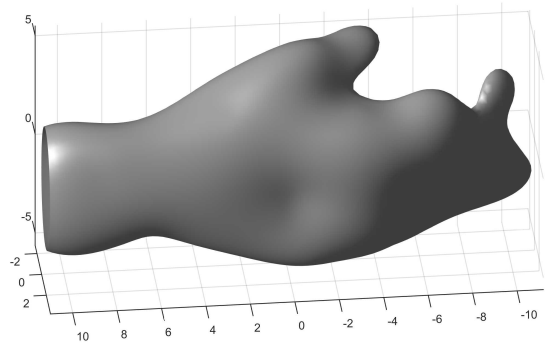
```
C = hrbf_centers(V, F, 20);
[P, N] = hrbf_points(V, F);
K = hrbf_coefficients(C, P, N);
```

Once this precomputation is done, distance and gradient can be evaluated for any point in space using `hrbf_apply`.

```
[D, G] = hrbf_apply(C, K, ...
  [1.0, 0.2, 0.5]);
```

As stated before, most functions support vectorized arguments. As such, multiple points can be queried at once.

```
[X, Y, Z] = meshgrid( ...
  -13.0 : 0.25 : 11.0, ...
  -5.0 : 0.25 : 5.0, ...
  -7.0 : 0.25 : 7.0);
D = hrbf_apply(C, K, ...
  [X(:), Y(:), Z(:)]);
D = reshape(D, size(X));
```



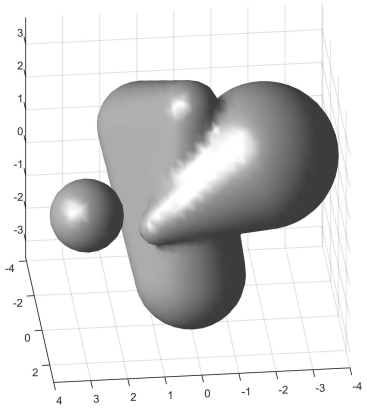
D.2 Spherical primitives

Spheres, bispheres and trispheres are respectively implemented in `sphere_single`, `sphere_double` and `sphere_triple`. In a similar fashion to `hrbf_apply`, these methods compute distances and gradients for given points.

```
[D, G] = sphere_single( ...
  [0.0, 0.0, 0.0], 1.0, ...
  [1.0, 2.0, 3.0]);
```

As spherical primitives are usually combined, `sphere_apply` takes an array of spheres and an array of indices.

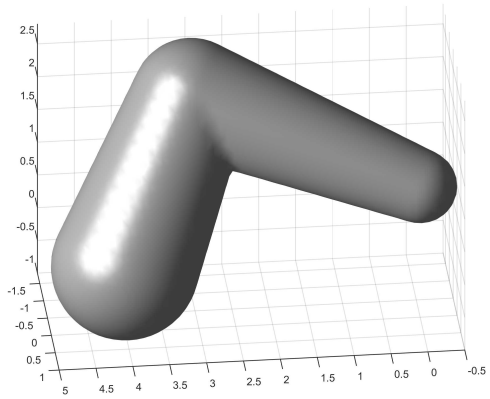
```
S = cell(6, 1);
S{1}.center = [3.0, 2.0, 1.0];
S{1}.radius = 1.0;
S{2}.center = [-2.0, 0.0, 1.5];
S{2}.radius = 2.0;
S{3}.center = [1.0, 1.0, 0.0];
S{3}.radius = 0.5;
S{4}.center = [0.0, 0.0, 3.0];
S{4}.radius = 0.8;
S{5}.center = [0.0, 0.0, -2.0];
S{5}.radius = 1.5;
S{6}.center = [1.0, -3.0, 1.0];
S{6}.radius = 1.2;
I = cell(3, 1);
I{1} = [1];
I{2} = [2, 3];
I{3} = [4, 5, 6];
[X, Y, Z] = meshgrid( ...
  -5.0 : 0.25 : 5.0, ...
  -5.0 : 0.25 : 5.0, ...
  -5.0 : 0.25 : 5.0);
D = sphere_apply(S, I, ...
  [X(:), Y(:), Z(:)]);
D = reshape(D, size(X));
```



D.3 Operators

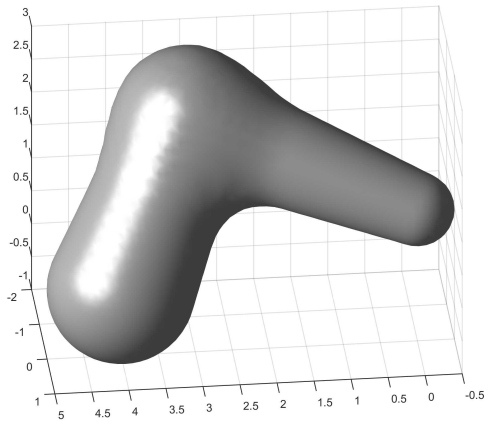
The function `sdf_reparam` converts global distance to local distance. The union operator is implemented in `sdf_union`.

```
[X, Y, Z] = meshgrid( ...
  -3.0 : 0.1 : 5.0, ...
  -2.0 : 0.1 : 5.0, ...
  -5.0 : 0.1 : 5.0);
D1 = sphere_double( ...
  [0.0, 0.0, 1.0], 0.5, ...
  [3.0, -1.0, 2.0], 0.8, ...
  [X(:), Y(:), Z(:)]);
D2 = sphere_double( ...
  [3.0, -1.0, 2.0], 0.8, ...
  [4.0, 0.0, 0.0], 1.0, ...
  [X(:), Y(:), Z(:)]);
D1 = sdf_reparam(D1, 2);
D2 = sdf_reparam(D2, 2);
D = sdf_union(D1, D2);
D = reshape(D, size(X));
```



Similarly, `sdf_blend` computes the blend operators, with the additional parameter θ. The function `std_blend_theta_curve` can be used to get θ from α, as suggested in section 3.4.2.

```
alpha = pi / 2;
theta = sdf_blend_theta_curve( ...
  alpha, ...
  pi / 8, pi / 2, pi / 2, ...
  pi / 4, pi / 10, pi / 10, ...
  1 / 2, 1 / 2);
D = sdf_blend(D1, D2, theta);
D = reshape(D, size(X));
```



As explained in section 3.4.2, our blend strategy is to set α as the angle between the two bones and interpolate between the union and the blend operators.

D.4 Bones and traditional skinning

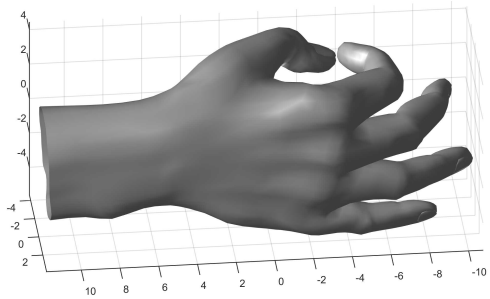
Since a lot of various data is required to represent a mesh, we packed everything in a structure. Vertices, faces, normals, skinning weights, HRBFs, spheres and some precomputations are stored in `neutral.mat`.

As input, our algorithm gets the spheres used in hand tracking. The method `bone_axes` computes expected bone transformations. Since the locations may not necessarily be consistent with our bone structure, the function `bone_transforms` provides rigid transforms by constraining bone lengths.

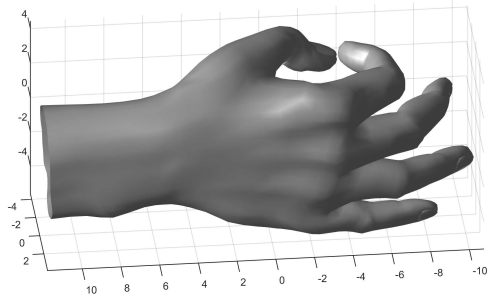
```
load('neutral.mat');
C = get_centers_somehow();
T = bone_transforms(mesh, C);
```

Once transforms are defined, `skin_linear` and `skin_dualquat` can be used to perform traditional skinning, as described in section 3.2.

```
lbs = skin_linear(mesh, T);
```

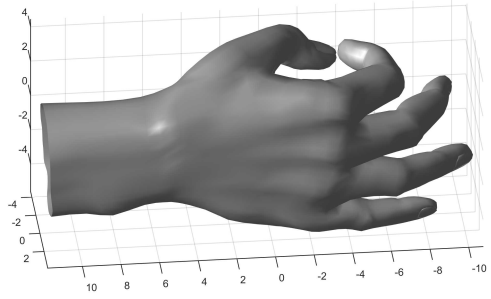


```
dqs = skin_dualquat(mesh, T);
```



Using a similar interface, implicit skinning can be also applied to a mesh. The method `skin_implicit` heavily relies on `sdf_field`, which computes distances and gradients for every vertex. As these two functions follows the algorithms described in this report in a straightforward manner, the code is self-explanatory.

```
is = skin_implicit(mesh, T, 2);
```



D.5 Tests

Finally, several scripts are provided, which are either demonstration code or preprocessing tools. We encourage the reader to go through some of them to understand how the functions can be used. Most of them were used to generate images of this report.

- `test_hrbf`, `test_sphere`, `test_operators` and `test_bulge` provide complete example of local and global supported functions.
- `test_2d_skinning`, `test_compare_params` and `test_compare_skinning` show differences between techniques and the impact of some parameters.
- `test_generate_neutral` was used to compute and group all data related to skinning into `neutral.mat`. `test_generate_hrbf` provides a better interface to tweak HRBF coefficients.
- `test_export` generates `neutral.obj` and `neutral.skl`, which are used in the C++ implementation.

E C++ code

In order to test actual real-time performances of implicit skinning, a C++ implementation was developed. Design goals are quite different from the MatLab version. Most of the preprocessing code has not been ported to C++, since actual data can be exported from MatLab directly. Moreover, some parts of the algorithm were modified to produce a simple yet functional code.

Since the goal is to provide real-time animation, some parts of the pipeline were simplified. The major simplification is that the blend is not supported. Only the union operator is implemented, mainly for time consideration, as the C++ implementation was achieved close to the deadline. Hence, no bulging effect is available.

Another simplification is the absence of explicit optimization. In order to have a clean and understandable code, only pure and straightforward C++ is used. Vaillant's implementation, available on his website, precomputes most of the distance functions and use *CUDA* to parallelize computations on GPU. Hence, even with a simpler algorithm, we do not expect to reach similar performances.

E.1 Skinning

For simplicity's sake, the skinning library is contained in a single header `skinning.h`.

Vector, matrices and dual quaternions are provided by *OpenGL Mathematics*[6], also known as *GLM*. This lightweight header-only library is designed to mimics *OpenGL Shading Language* data structures. Several extensions were added, which simplifies our code for dual quaternion skinning.

The other dependency is *OpenMesh 5.1*[5] to read and manipulate meshes. Its halfedge data structure provides efficient and intuitive access to vertices neighbors.

The `HRBF` structure provides signed distance function for Hermite RBF. While this is the only one implemented, any structure with a similar inter-

face would work seamlessly with our library. More precisely, any object with the following method is compatible:

```
float apply(glm::vec3 const & point,
  glm::vec3 & gradient) const;
```

As explained above, the C++ code does not provide any precomputation. As such, coefficients and centers must be provided, for instance using our MatLab implementation.

We also define the type `Mesh`, which is an OpenMesh triangular mesh. As long as normals are available, any mesh should work with our library. The only requirement is the single precision for floating point numbers. Our code assumes that `glm::vec3` and `Mesh::Point` are binary equivalents.

The main structure is `Implicit`, which defines linear blend, dual quaternion and implicit skinning. Given a mesh with skinning weights and desired bone transforms, geometry can be deformed with one of those algorithms:

```
// Import HRBF somehow
Implicit<HRBF> implicit;
implicit.sdfs = /* ... */;

// Load mesh using OpenMesh
implicit.mesh = /* ... */;

// Import weights somehow
implicit.weights = /* ... */;
implicit.weights_bone = /* ... */;

// Initialize internal data
implicit.initialize();

// Ready to skin
implicit.transforms = /* ... */;
implicit.skin_linear();
implicit.skin_dualquat();
implicit.skin_implicit();
```

E.2 Demo

A minimal application is available in `demo.cpp`, using *OpenGL* to render the mesh. *OpenGL Extension Wrangler*[10], also known as *GLEW*, is used to load *OpenGL functions*. The context is created in a portable way with *GLFW 3*[7].

Most of the code is used to set up the OpenGL context and render the scene. As this is not the subject of this document, we will not discuss low-level rendering. Anyway, the code was designed to have minimal performance penalty.

The mesh is loaded from `neutral.obj`, which is a regular *Wavefront OBJ file*. Weights and HRBF coefficients are read from `neutral.skl`, which is a custom file using an OBJ-like text-based encoding. The `test_export` script in MatLab generate these files from `neutral.mat`.

While the actual performances depends on hardware configuration, linear blend and dual quaternion skinning have similar efficiencies. Implicit skinning is about 5 times slower, with approximately 100 frames per seconds on our configuration. Our computer has an *Intel Core i7-4700HQ* which runs at 2.40GHz. However, as we explained before, the code is not optimized and could benefits from hardware parallelization. On the other hand, the bulging effect is not implemented, which makes the code lighter.

References

[1] Ilya Baran and Jovan Popović. "Automatic Rigging and Animation of 3D Characters". In: *ACM Trans. Graph.* 26.3 (July 2007). ISSN: 0730-0301. DOI: 10.1145/1276377.1276467. URL: http://doi.acm.org/10.1145/1276377.1276467.

[2] Loïc Barthe, Brian Wyvill, and Erwin de Groot. "Controllable Binary Csg Operators for "soft Objects"". In: *International Journal of Shape Modeling* 10.2 (2004), pp. 135–154. DOI: 10.1142/S021865430400064X. URL: http://dx.doi.org/10.1142/S021865430400064X.

[3] Sofien Bouaziz and Mark Pauly. "Dynamic 2D/3D Registration for the Kinect". In: *ACM SIGGRAPH 2013 Courses*. SIGGRAPH '13. Anaheim, California: ACM, 2013, 21:1–21:14. ISBN: 978-1-4503-2339-0. DOI: 10.1145/2504435.2504456. URL: http://doi.acm.org/10.1145/2504435.2504456.

[4] Sofien Bouaziz, Andrea Tagliasacchi, and Mark Pauly. "Sparse Iterative Closest Point". In: *Proceedings of the Eleventh Eurographics/ACMSIGGRAPH Symposium on Geometry Processing*. SGP '13. Genova, Italy: Eurographics Association, 2013, pp. 113–123. DOI: 10.1111/cgf.12178. URL: http://dx.doi.org/10.1111/cgf.12178.

[5] Department of Computer Graphics and Multimedia of RWTH-Aachen University. *OpenMesh 5.1*. URL: http://www.openmesh.org/.

[6] G-Truc Creation. *GLM: OpenGL Mathematics*. URL: http://glm.g-truc.net/.

[7] GLFW development team. *GLFW3*. URL: http://www.glfw.org/.

[8] Olivier Gourmel et al. "A Gradient-based Implicit Blend". In: *ACM Trans. Graph.* 32.2 (Apr. 2013), 12:1–12:12. ISSN: 0730-0301. DOI: 10.1145/2451236.2451238. URL: http://doi.acm.org/10.1145/2451236.2451238.

[9] Kai Hormann and Michael S. Floater. "Mean Value Coordinates for Arbitrary Planar Polygons". In: *ACM Trans. Graph.* 25.4 (Oct. 2006), pp. 1424–1441. ISSN: 0730-0301. DOI: 10.1145/1183287.1183295. URL: http://doi.acm.org/10.1145/1183287.1183295.

[10] Milan Ikits and Marcelo Magallon. *GLEW: OpenGL Extension Wrangler*. URL: http://glew.sourceforge.net/.

[11] Alec Jacobson et al. *gptoolbox: Geometry Processing Toolbox*. 2015. URL: http://github.com/alecjacobson/gptoolbox.

[12] Ladislav Kavan et al. "Geometric Skinning with Approximate Dual Quaternion Blending". In: *ACM Trans. Graph.* 27.4 (Nov. 2008), 105:1–105:23. ISSN: 0730-0301. DOI: 10.1145/1409625.1409627. URL: http://doi.acm.org/10.1145/1409625.1409627.

[13] Ives Macêdo, João Paulo Gois, and Luiz Velho. "Hermite Interpolation of Implicit Surfaces with Radial Basis Functions". In: *Proceedings of the 2009 XXII Brazilian Symposium on Computer Graphics and Image Processing*. SIBGRAPI '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–8. ISBN: 978-0-7695-3813-6. DOI: 10.1109/SIBGRAPI.2009.11. URL: http://dx.doi.org/10.1109/SIBGRAPI.2009.11.

[14] Rodolphe Vaillant et al. "Implicit Skinning: Real-time Skin Deformation with Contact Modeling". In: *ACM Trans. Graph.* 32.4 (July 2013), 125:1–125:12. ISSN: 0730-0301. DOI: 10.1145/2461912.2461960. URL: http://doi.acm.org/10.1145/2461912.2461960.