



Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Improving Delta Mush Based Character Animation

Scientific Students' Association Report

Author:

Gergő Viktor Haragos

Advisor:

dr. Márton Vaitkus

2024

Contents

Abstract	1
1 Introduction	2
2 Background and related work	4
2.1 Skeletal Animation	4
2.2 Skinning	5
2.3 Linear Blend Skinning	5
2.4 Finding skinning weights	7
2.5 Handling self-intersections, implicit skinning	9
3 Delta Mush	11
3.1 Overview	11
3.2 Laplacian smoothing	13
3.3 Computing and restoring delta vectors	13
3.4 Limitations and problem statement	15
4 Proposed Method	16
4.1 Overview	16
4.2 Continuous Collision Detection	17
4.3 Delta Mush with CCD	18
4.4 Improving mesh quality	20
4.5 Improving performance	21
5 Results	22
5.1 Deformation examples	22
5.2 Performance	27
6 Conclusion	29
6.1 Summary	29
6.2 Future work	29

Acknowledgements	30
Bibliography	31

Abstract

Character animation is one of the classical problems of computer graphics. In the traditional animation process characters are modeled as polygonal meshes, animated using via a suitable skeleton structure. In the de facto standard method known as linear blend skinning, the transformations of each bone are transferred onto the skin of the character using a weighted average [14]. While some methods exist for the automatic generation of such skinning weights, in current practice they still require careful manual tuning by artists. A promising alternative to manual weight painting is the “Delta Mush” method [19]. The basic steps are: 1) animate the model using the simplest possible skinning weights, 2) strongly smooth the mesh (“mush”) to get rid of the artifacts of the imperfect weighting, and finally 3) restore surface details by adding local displacement vectors (“deltas”) to the smoothed mesh vertices.

While the Delta Mush method is straightforward to implement and is also computationally efficient [16], it doesn’t prevent self-intersection of the skinned surface. In this work my goal is to investigate how this shortcoming could potentially be remedied. I describe a method which detects self-intersections using continuous collision detection techniques and modifies the delta vectors to ensure a valid result. The work also discuss variations of the original delta mush method, and how it can be integrated with common animation tools such as blendshapes.

Chapter 1

Introduction

Animation of scenes and their virtual characters is a fundamental element in the development of filmmaking, special effects and computer games. In traditional animation, artists had to draw the movements frame by frame, which is a time-consuming process. However, with advances in computer technology, artists can increasingly rely on computer-generated imagery (CGI), which can be used to create almost any kind of moving image.

Computer-generated animation is the virtual equivalent of "stop motion", where temporal movement is achieved by placing animated characters in different poses frame by frame. However, such a manual procedure would be quite labor-intensive, so in practice usually only discrete keyframes are defined, and in-between frames are automatically generated using interpolation methods. The keyframes are placed on a virtual timeline, where the dynamics of the movement can be further refined by adjusting the temporal spacing to speed up or slow down the movement.

Three-dimensional characters are typically modeled using polygonal surface meshes. A mesh could consist of thousands or even millions of elements, so directly specifying the precise movement of each point of the surface would simply be unfeasible. In practice, animations are defined by specifying affine transformations for the elements (bones) of some kind of skeleton structure that are combined to determine the movement of the surface mesh [20]. The correspondence between the skeleton and the mesh is created in the process known as skinning, for which the most popular technique is to simply combine the transformations of each bone using weighting functions defined over the mesh. While many methods exist for the automatic computation of such skinning weights [13], in practice artists generally have to rely on extensive manual tuning (weight painting) to achieve the desired look.

A recently developed approach that has the promise of circumventing the need for manual weight painting is known as Delta Mush (DM) [19]. This technique is based on the observation that even very simple skinning methods (e.g. using nearest-neighbor weights) are acceptable over most of the model, with problems arising only in certain regions in the form of sharp turns. Getting rid of such artifacts is possible by simply smoothing the geometry of the transformed mesh (mush). Such smoothing however will also destroy high-frequency details, which need to be restored in the form of displacement vectors (deltas).

While delta mush alleviates some of the difficulties associated with the traditional animation workflow, and can even be made suitable for real-time computation [16], it also has some well-known limitations. In particular, just like most other skinning methods, delta mush does not prevent the self-intersection of the mesh. The starting point of my work is

the observation that the delta vector field playing a key role in the Delta Mush approach could also be used to handle self-intersections in a natural way.

The main subject of my thesis is a method for resolving self-intersections during Delta Mush based animations using the methods of Continuous Collision Detection (CCD). In Chapter 2, an overview of the relevant technical background and related work is given, followed by a detailed description of Delta Mush in Chapter 3. Then, in Chapter 4, I present my proposed method for resolving self-intersections and present some practical results in Chapter 5. Finally, the report concludes with an overview of various avenues of future research.

Chapter 2

Background and related work

2.1 Skeletal Animation

Characters moving in a virtual 3D scene are usually represented by textured triangular meshes, called meshes or skins [20] – see Figure 2.1. Computer animation of a character therefore requires the trajectory of the corresponding mesh vertices to be specified (at least in keyframes). A typical mesh may consist of thousands or even millions of triangles, so moving all vertices independently is not practical. In practice, some kind of control structure is used to animate the characters, in the case of animal or humanoid characters specifically some kind of skeleton.

Skeletal animation is a technique in computer animation in which a character (or other articulated object) is made up of two components: the surface mesh used to represent the character, and a hierarchical set of interconnected parts (called bones) that together form a skeleton (or skeleton-rig), a virtual armature that can be used to animate the mesh (i.e. define poses and keyframes). While this technique is typically used to animate humans and other organic figures, it is primarily used to make the animation process more intuitive, and the same technique can be used to control the deformation of any object. If the object being animated is more general the set of "bones" may not be hierarchical or coherent, but simply a higher-level description of the movement of the part of the mesh it affects. The default pose of a humanoid 3D model skeleton before animating it is called the T-pose (or sometimes A-pose), which is used by most 3D software as the starting position to which transformations are later applied to produce various poses.

The skeleton is defined as a series of bones (which do not have to correspond to real-world anatomical features). During animation each bone is assigned a three-dimensional transformation that maps from the default spatial configuration (which includes its position, size and orientation) to the current configuration. The bones generally form a tree-like hierarchy, with each bone having a unique parent (although this is not always the case). The complete transformation of a given bone is the composition of its parent transformation and its own transformation. So, for example, moving a thigh bone will cause the corresponding lower leg to move. In skeletal animation, an artist specifies the movement of individual bones. The character model must naturally follow the movement of the skeleton – this is achieved by the process referred to as skinning.



Figure 2.1: Source: author’s own work

2.2 Skinning

Skinning is one of the most important techniques in 3D animation [14], allowing a mesh to be linked to a skeleton so that the mesh deformation naturally follows the movement of the skeleton, which is particularly important for character animation. Skinning methods used in practice, such as Linear Blend Skinning (described later), apply weighted combinations of affine transformations (e.g., displacement, rotation) to surface points. Each bone is assigned a corresponding weight function (e.g., skinning weight), determining the influence of the bone’s movement on each vertex of the mesh. The weights are often represented to the user by color-coding: each bone has an assigned color, and weights are visualized through color blending – an example is shown in Figure 2.2. However, setting optimal weights is challenging. The available methods may not be sufficient, often requiring manual intervention to refine the weights for the desired effect. In most state-of-the-art graphics engines, the skinning process is performed on the GPU within a shader program. In the rest of this section, I will describe various skinning techniques implemented during my research.

2.3 Linear Blend Skinning

Linear blend skinning (LBS), also known as skeletal subspace deformation is one of the most widely used algorithm for models of skeleton-based direct deformation [14]. Some of the early ideas were already presented in pioneering works [1] and [11]. The formal description of the method probably goes back to Lewis et al [10], but by then the algorithm was already widely known and commercially implemented in commercial software packages. LBS is based on the following input data:

- Resting pose shape, typically represented as a polygon mesh with vertex positions $\mathbf{v}_1 \dots, \mathbf{v}_n \in \mathbb{R}^4$ (using homogeneous coordinates). It is assumed that the topology of the mesh is fixed and only the positions of the vertices change during deformation.
- A list of bone transformations, given as matrices $\mathbf{T}_1, \dots, \mathbf{T}_m \in R^{4 \times 4}$.
- Skinning weights for each bone and each vertex: $w_{i,1}, \dots, w_{i,m} \in \mathbb{R}, i = 1, \dots, n$. Each weight $w_{i,j}$ gives the influence of bone j on vertex i . As a general requirement, the

weights form a convex partition of unity, i.e. they lie between 0 and 1, and their sum at each vertex equals to 1.

Using linear blend skinning the deformed vertex positions are calculated according to the following formula:

$$v'_i = \sum_{j=1}^m w_{i,j} \mathbf{T}_j \mathbf{v}_i = \left(\sum_{j=1}^m w_{i,j} \mathbf{T}_j \right) \mathbf{v}_i \quad (2.1)$$

In simple terms the rest pose vertices are transformed by a weighted linear combination (blend) of bone deformations.

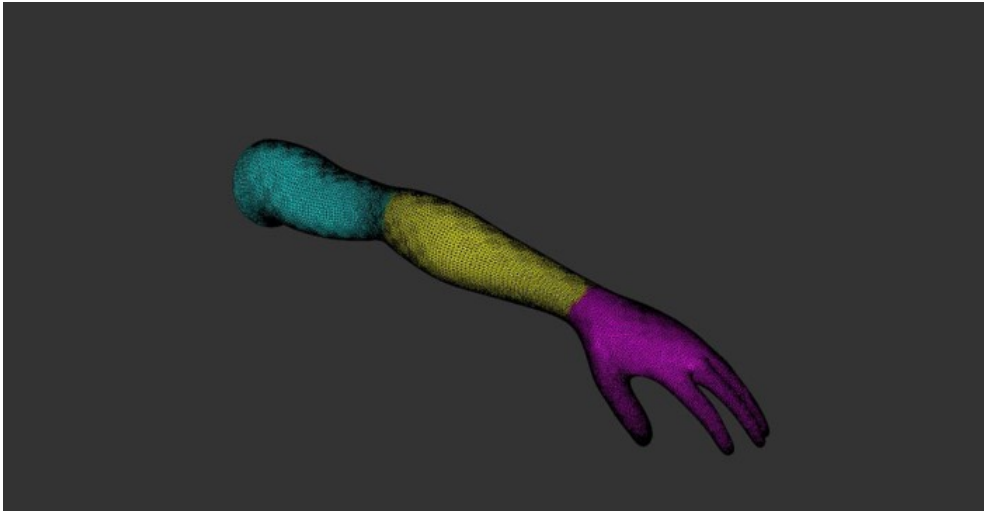


Figure 2.2: Visualization of skinning weights.

Although arbitrary affine transformations are allowed, it is usually convenient to assume that each transformation is a rigid body (Euclidean) transformation, i.e. that $\mathbf{T}_j \in SE(3)$.

While linear blend skinning is widely used due to its computational efficiency, it suffers from various shortcomings. One well-known class of artifact arises because transformations are combined using simple linear combinations, ignoring the nonlinear nature of rotations. In particular the average of two opposite rotations could produce a zero transformation, thus producing the famous candy-wrapper artifact – see Figure 2.3. Many alternatives to LBS have been proposed [14] with the goal of mitigating these problems, such as Dual Quaternion Skinning [15], which defines a more complex nonlinear blending of Euclidean transformations. LBS and many of its alternatives employ weight functions – the determination of these is a major issue in practice that we discuss next.

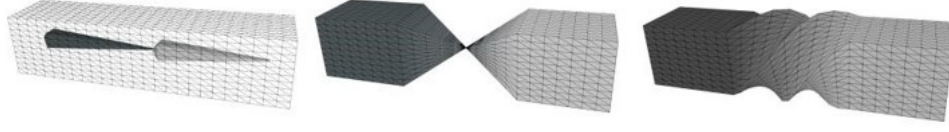


Figure 2.3: Rest pose (left), linear blend skinning with candy-wrapper artifact (center), dual quaternion skinning (right). Source: [14].

2.4 Finding skinning weights

As we saw in the previous section, linear blend skinning requires a suitable system of skinning weight functions. Determining the weight distribution often involves manual fine-tuning to achieve the desired effect, but it is useful to have a default weight distribution with the following properties:

1. The weights form a convex combination (see previous chapter),
2. The distribution of weights along the surface is continuous (at least C^0), smooth (at least C^1), and uniformly distributed,
3. The size of the weights decreases as we move further away from the bones that correspond to them.

A very simple way of obtaining the weight distribution is as follows: if the nearest bone to a given point is of index 1, then its weight is 1. This weighting satisfies properties 1) and 3) trivially, but not property 2), since the weights change in a discontinuous manner [13]. A slightly more sophisticated method is inverse-distance weighting, where the weight of each surface point is some power of the reciprocal of the distance from each bone:

$$w_{i,j} = \frac{\frac{1}{d(v_i, b_j)^p}}{\sum_k \frac{1}{d(v_i, b_k)^p}}. \quad (2.2)$$

This weighting, although it fulfills all three desired properties, is not very practical as it does not account for 3D points on different parts of the surface that may be at similar distances from the bones.

Baran and Popovic [2] proposed an alternative, two-step approach to automatically calculate skeletal weights. Their method is often referred to as "Bone Heat" because it is based on the physical analogy of heat diffusion. The uniform weight distribution can be determined by optimizing the energy:

$$E(w_k) = \int_M \left((\nabla w_k(x))^2 + \lambda \frac{1}{d_k(x)^2} (w_k(x) - \bar{w}_k(x))^2 \right) dM \rightarrow \min, \quad (2.3)$$

where w is some default weight distribution (e.g. 0 or 1) and $\bar{w}$ is the smoothed weight distribution we want to determine, λ is a weighting factor balancing smoothness and

preservation of the original weights, and $d_k(x)$ is the distance from the nearest k indexed bone. Differentiating this energy and equating to 0, we obtain:

$$\left(\Delta + \lambda \frac{1}{d_k(x)^2}\right) w_k(x) = \lambda \frac{1}{d_k(x)^2} \bar{w}_k(x), \quad (2.4)$$

where Δ is the Laplace operator, expressing deviation from environmental average. Discretizing this differential equation over the mesh, we get:

$$(\mathbf{L} + \mathbf{D})\mathbf{w} = \mathbf{D}\bar{\mathbf{w}}, \quad (2.5)$$

$\mathbf{D}$ is a diagonal matrix with nonzero elements only along the principal diagonal:

$$(\mathbf{D})_{i,i} = \lambda \frac{1}{d_k(x_i)^2}. \quad (2.6)$$

$\mathbf{L}$ is the so-called discrete ("cotangent") Laplace matrix, whose elements in row i and column j are not 0 if i and j are connected by edges in the mesh and then the element is given by the following formula [5]:

$$(\mathbf{L})_{ij} = -\frac{\cot \theta_{ij} + \cot \theta_{ji}}{2}, \quad (2.7)$$

and the diagonal elements are equal to minus the sum of off-diagonal elements in the same row.

Solving the above linear system of equations a highly smooth set of weights can be generated that can be tuned by the simple scalar weight α . An example of the weights generated by my own implementation of the bone heat method is shown in Figure 2.4.

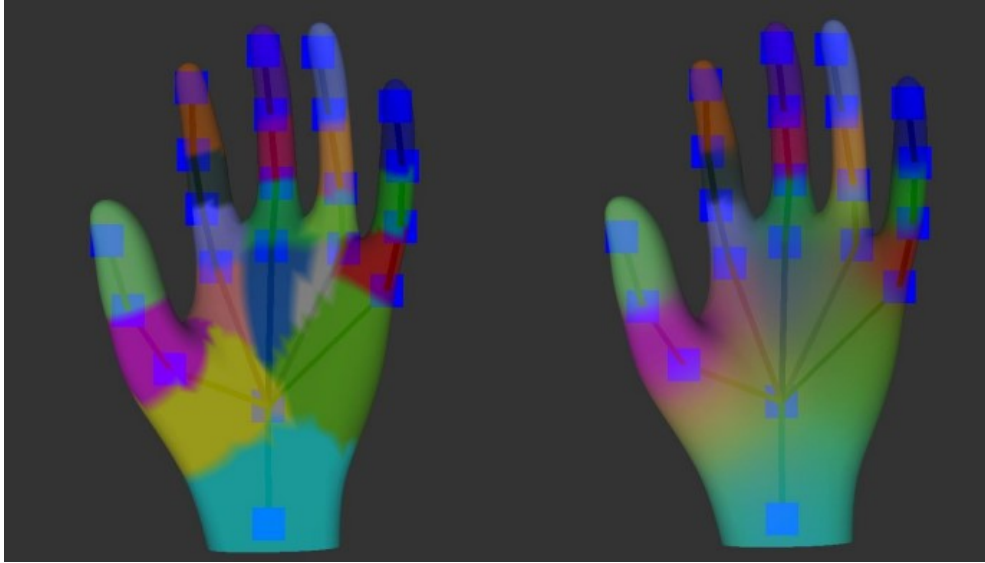


Figure 2.4: Original nearest-neighbor weights (left), bone heat weights (right).

While bone heat often generates acceptable weights, manual weight painting might still be necessary to produce acceptable animations. A recently proposed alternative approach that promises to greatly simplify the skinning process is the Delta Mush method [19], which is described in great detail in the next Chapter.

2.5 Handling self-intersections, implicit skinning

Using most of the standard skinning-based techniques it could easily occur that the animated mesh intersects itself, which must be corrected (at least when the self-intersection is visually noticable). One possibility is to go beyond simple surface mesh animation, and represent the interior volume of the characters as well, which might even include the physically-based simulation of a model of the skeletal and muscular structure of the characters. Such advanced techniques are occasionally used for the visual effects of high-budget motion pictures, but their application currently requires great expertise and significant resources.

The common practice of most of the animation industry is to apply so-called corrective blendshapes [17], that are manually created displacements of the mesh which resolve self-intersection or other animation artifacts. The creation of such corrective shapes (also called sculpts [7]) is supported by commonly used animation tools and is part of the common skill-set of professional animators. Nevertheless, the manual nature of the process motivates the search for methods that could automatically resolve or even prevent self-intersections.

One method with such capability is implicit skinning, introduced by Valliant et al. [23]. The basic idea behind implicit skinning is to represent the animated surface as the level set of an implicit function, which mathematically guarantees that self-intersection will not occur in any configuration. To calculate the implicit surface, the mesh is first decomposed into disjoint submeshes, each corresponding to one of the bones of the skeleton rig. Then, the Euclidean distance field of each of the submeshes is approximated with a combination of radial basis functions:

$$d_i(\mathbf{x}) = \sum_{k=1}^m \left(\lambda_k \varphi(\|\mathbf{x} - \mathbf{v}_k\|) + \beta_k^T \nabla \varphi(\|\mathbf{x} - \mathbf{v}_k\|) \right) \quad (2.8)$$

where v_k are carefully chosen sample points and m is the number of samples. The vectors λ_k and β_k are the unknown coefficients and φ is a radial basis function, such as $\varphi(r) = r^3$. The coefficients are determined by solving Hermite interpolation problem, i.e. a linear system of constraints on field values $d_i(v_k) = 0$, as well as gradients $\nabla d_i(v_k) = n_k$.

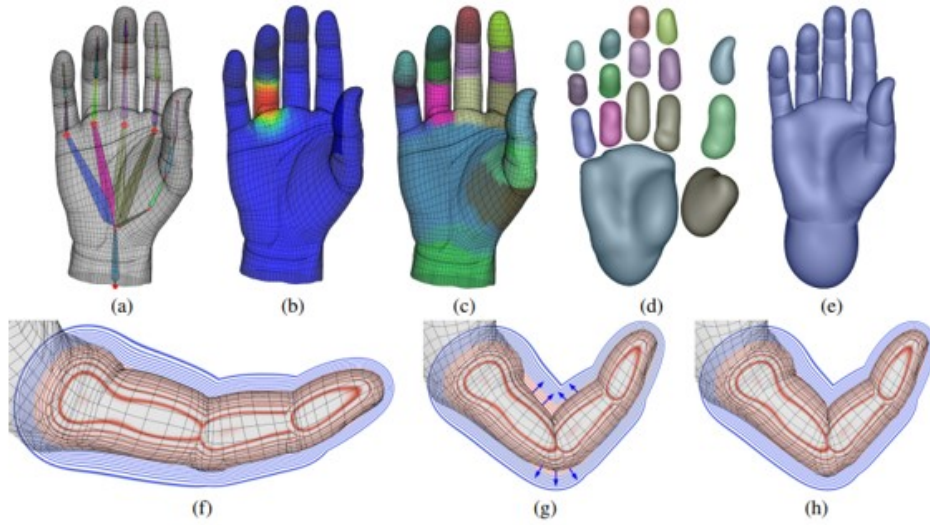


Figure 2.5: Illustration of implicit skinning. Taken from [23]

This calculation of the implicit functions is carried out only for the rest pose of the mesh, while during animation the distance fields are simply transformed in space according to the bone transformations. The transformed partial distance fields are then blended together to produce an implicit representation of the original mesh. For this purpose, various gradient-based implicit blending operators have been proposed in the original paper [23], and some follow-up works [24].

The level surface of this implicit function is then guaranteed to be free of any self-intersection. Although this level surface could be extracted in the standard way using e.g. the method of marching cubes [18], instead the skinned mesh is displaced so that its vertices lie on the same level set as in the rest pose. While implicit skinning remains an attractive alternative to traditional methods, it is also known to be computationally expensive.

In our work we wanted to investigate an alternative approach to prevent self-intersection when applying the Delta Mush method. For this purpose we employed the techniques of Continuous collision Detection (CCD), which will be the topic of the following Chapters.

Chapter 3

Delta Mush

3.1 Overview

Delta Mush is a method first proposed by Mancewicz et al. [19] and later improved and analyzed by Le and Lewis [16], that provides a solution some of the long-standing challenges associated with skeleton-based mesh animation. It has some important advantages over more traditional methods, as it produces acceptable animations without carefully constructed skinning weights – in fact it only needs the most basic kind of weighting based on e.g. nearest neighbor interpolation (see Chapter 2). The key observation motivating this approach is that when animating with such simple weights the result mostly has the desired shape, with the exception of some high-frequency artifacts (e.g. sharp edges) arising in the vicinity of skeleton joints. If we then apply sufficiently strong smoothing to the deformed mesh these artifacts can be made to disappear entirely.

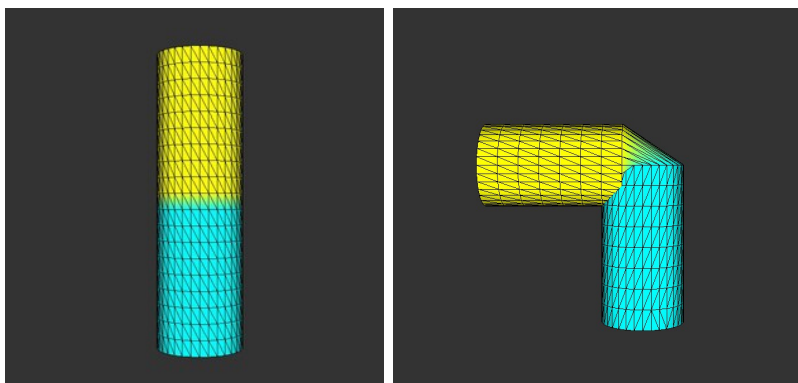


Figure 3.1: Rest pose (left) and LBS deformation with simple 0-1 weighting (right).

The key operation of Delta Mush is thus computing a smoothed version of the mesh (called the *mush*), both in the initial resting position (T-pose) and the current deformed pose. While the mush for the deformed pose should be free of artifacts, it will lose various geometric details compared to the original surface. Thus we compute difference vectors (*deltas*) between the *original* mesh and its mushed version with respect to a vertex-based local coordinate system – see Figure 3.2 for an illustration. By adding the same delta vectors to the mush of the *deformed* mesh the lost details can be restored and we arrive at an artifact-free deformed mesh. The steps of this process are shown in Figure 3.3.

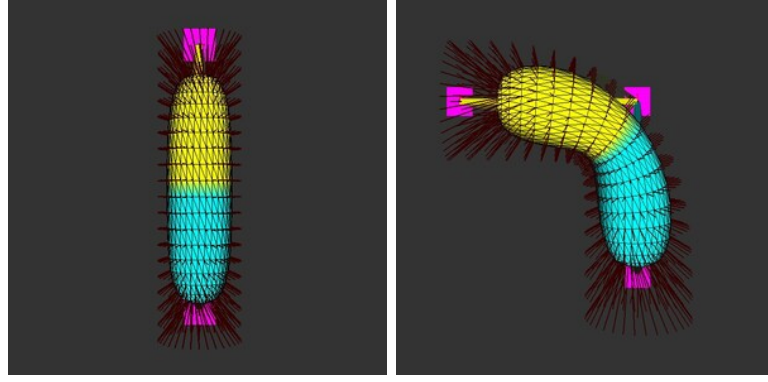


Figure 3.2: Illustration of the smoothed mesh and its delta vectors in the rest pose (left) and the deformed pose (right).

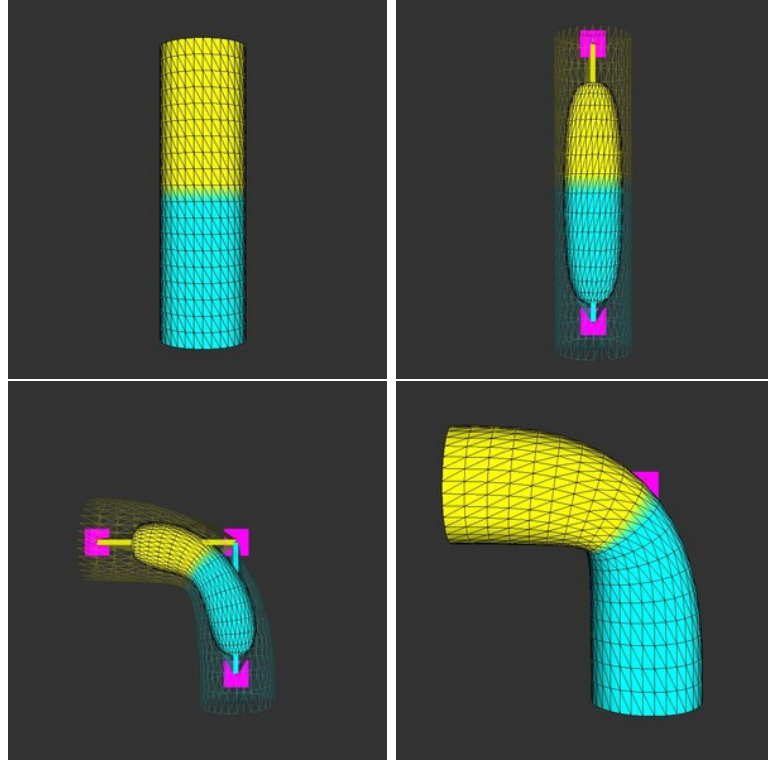


Figure 3.3: Illustration of the delta mush method: original rest pose mesh (top left), smoothed rest pose mesh (top right), smoothed deformed mesh (bottom left), and final deformation (bottom right).

Using the Delta Mush method reasonably high quality skeleton-based deformations can be computed without manual weight painting or complex automatic weight computations. Delta Mush be seen as an adaptation of the various displacement-based and multiresolution methods commonly used for the deformation and modeling of polygonal meshes [3, 5]. In the rest of this Chapter, we describe in more detail the most important elements of this approach, namely Laplacian smoothing and the computation and transformation of delta vectors, and discuss some of the limitations of Delta Mush, that served as a starting point of the current work.

3.2 Laplacian smoothing

A crucial element of Delta Mush is the smoothing of the mesh. For this purpose, usually Laplacian smoothing [5] is employed, which is a commonly used technique in polygonal mesh processing. The basic idea is very simple: each mesh vertex will be some weighted average of its immediate neighbours. The simplest version of Laplacian smoothing simply moves each vertex to its neighbours:

$$\mathbf{p}'_i = \sum_{j=0}^N w_j \mathbf{p}_j, \quad (3.1)$$

where $\mathbf{p}_i$ and $\mathbf{p}'_i$ denote the original and smoothed positions of vertices respectively, while w_i are scalar weights – popular choices include uniform weights and cotangent weights (same as in e.g. bone heat, see Chapter 2).

To illuminate the reason for the name "Laplacian", let us rewrite the above formula

$$\mathbf{p}'_i = \mathbf{p}_i - (\mathbf{p}_i - \sum_{j=0}^N w_j \mathbf{p}_j), \quad (3.2)$$

which can be expressed in matrix form as

$$\mathbf{P}' = (\mathbf{I} - \mathbf{L})\mathbf{P}, \quad (3.3)$$

where, $\mathbf{P}, \mathbf{P}'$ collect the coordinates of every vertex of the mesh, $\mathbf{I}$ is the identity, and $\mathbf{L}$ is called a Laplacian matrix. A single explicit smoothing operation does not result in every vertex positioned at the average of its neighbors, but can be iterated multiple times:

$$\mathbf{P}' = (\mathbf{I} - \mathbf{L})^n \mathbf{P}. \quad (3.4)$$

In our experiments we found $n = 25$ iterations of explicit smoothing with cotangent weights was sufficient to resolve skinning artifacts.

A commonly used variation called implicit Laplacian smoothing [8] is defined as follows:

$$\mathbf{P}' = (\mathbf{I} + \mathbf{L})^{-1} \mathbf{P}. \quad (3.5)$$

Implicit smoothing can substitute multiple iterations of explicit smoothing at the cost of solving a linear system of equations, i.e. inverting the large (but highly sparse) matrix $\mathbf{I} + \mathbf{L}$. In this work, we chose not to utilize implicit smoothing.

Note that whether explicit or implicit Laplacian smoothing is used, the vertices of the mesh can be expressed as linear combinations of the original vertices – this observation is the basis of highly efficient implementations of Delta Mush, see [16].

3.3 Computing and restoring delta vectors

After Laplacian smoothing the mesh loses not only the various animation artifacts, but also many important geometric details that need to be restored, which is possible with the use displacement vectors called deltas.

First a coordinate system is created at each point of the smoothed rest pose mesh, based on the normal, tangent and bi-tangent vectors. The delta vector is calculated simply

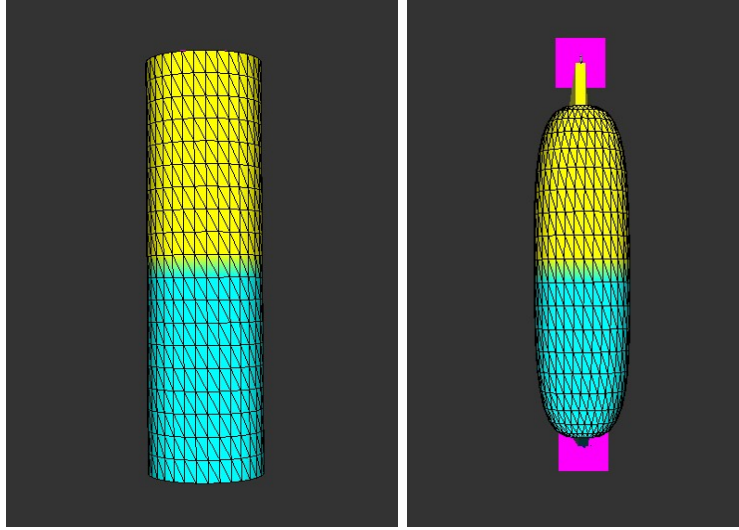


Figure 3.4: Mesh before and after Laplacian smoothing.

as the coordinates of the vertex on the unsmoothed mesh, expressed in this coordinate system, with the corresponding vertex on the smoothed mesh chosen as the origin. This can be written in an elegant and compact form using homogeneous coordinate vectors – see Algorithm 1 for detailed pseudocode.

Later, after the mesh has been smoothed in a deformed pose, the same delta vectors are added as displacements to restore details. Recall that the vectors are expressed in a local system, which is now changed to the normal, tangent and bi-tangent vectors of the deformed mesh (after smoothing). See Algorithm 2 for detailed pseudocode, again using a homogeneous coordinate representation.

Algorithm 1 Delta Mush: Calculate Delta Vector

```

1: function CALCULATEDELTA(M)
2:    $S \leftarrow \text{smooth}(M)$  ▷ Create smoothed mesh
3:    $D \leftarrow \text{init}(M)$  ▷ Initialize delta vector list
4:   for  $i \in [0, nVertex(M))$  do
5:      $p_i \leftarrow \text{getPoint}(M, i)$  ▷ Get point of original mesh
6:      $s_i \leftarrow \text{getPoint}(S, i)$  ▷ Get point of smoothed mesh
7:      $n_i \leftarrow \text{getNormal}(S, i)$  ▷ Get normal vector of smoothed mesh
8:      $t_i \leftarrow \text{calcTangent}(S, i)$  ▷ Calculate tangent vector of smoothed mesh
9:      $b_i \leftarrow \frac{t_i \times n_i}{\|t_i \times n_i\|}$  ▷ calculate bi-tangent vector of smoothed mesh
10:     $R_i \leftarrow [t_i, n_i, b_i, s_i]$  ▷ Create transformation matrix
11:     $D_i \leftarrow R_i^{-1} * p_i$  ▷ Store transformed point as delta
12:  end for
13:  return D
14: end function

```

Algorithm 2 Delta Mush: Calculate Final Deformation

```
1: function CALCULATEDEFORMATION(M,D)
2:    $S \leftarrow \text{smooth}(M)$  ▷ Create smoothed mesh
3:    $P \leftarrow \text{init}(M)$  ▷ initialize deformed point list
4:   for  $i \in [0, nVertex(M))$  do
5:      $s'_i \leftarrow \text{getPoint}(S, i)$  ▷ Get point of smoothed mesh
6:      $n'_i \leftarrow \text{getNormal}(S, i)$  ▷ Get normal of smoothed mesh
7:      $t'_i \leftarrow \text{calcTangent}(S, i)$  ▷ Calculate tangent vector of smoothed mesh
8:      $b'_i \leftarrow \frac{t_i \times n_i}{\|t_i \times n_i\|}$  ▷ Calculate bi-tangent vector of smoothed mesh
9:      $C_i \leftarrow [t'_i, n'_i, b'_i, s'_i]$  ▷ Create transformation matrix
10:     $P_i \leftarrow C_i * D_i$  ▷ Add transformed deltas to smoothed mesh
11:  end for
12:   $\text{SetMeshVertices}(M, P)$ 
13: end function
```

3.4 Limitations and problem statement

While the Delta Mush method has the potential of greatly simplifying traditional animation workflows is also has some limitations. First, it is markedly less efficient than direct linear blend skinning, due to the additional smoothing and delta calculations. This has been addressed in the work of Le and Lewis [16], who analyzed Delta Mush in great detail, and described a very efficient "direct" approach for its implementation, that even reached real-time performance. It was also discovered that the original Delta Mush is part of a larger family of methods, including many other variants that offer different trade-offs between quality and performance.

Another important limitation is that while Delta Mush can successfully remove certain geometric artifacts, it does not prevent self-intersections, which require many hours of manual labor to resolve in practice. Solving this problem for Delta Mush has been the main objective of this research project, and is the subject of the rest of the report.

Chapter 4

Proposed Method

4.1 Overview

As discussed in previous chapters, the main goal of this research project is to investigate methods for preventing the self-intersection of meshes when they are animated using a Delta Mush approach. Our approach is based on the observation that displacing the smoothed mesh with the delta vectors can be interpreted as moving the smoothed mesh through space, with the delta vectors prescribing a velocity for each vertex. When the mesh collides with itself during its movement, that signals a potential self-intersection at the final configuration. Thus, if neither the original, nor the deformed-then-smoothed mesh contain self-intersections, and the movement of mesh elements is terminated right before their collision, the displaced mesh must also be free of self-intersections.

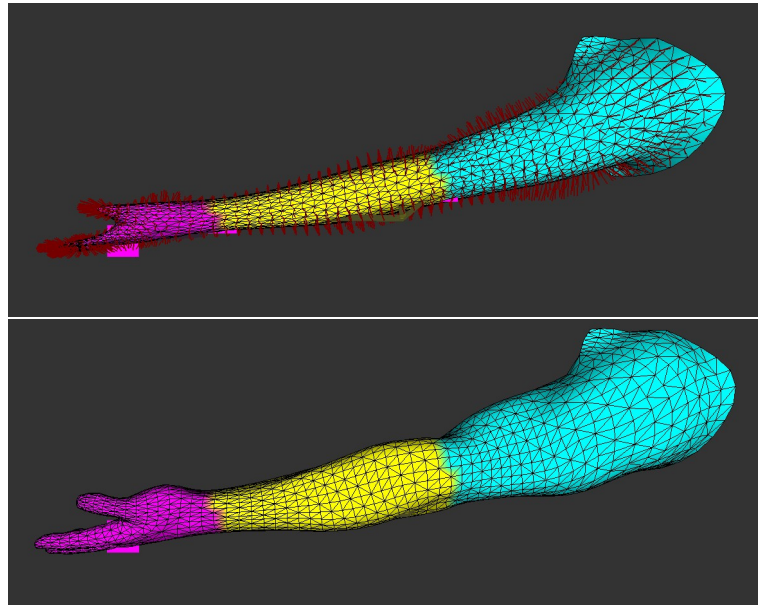


Figure 4.1: Illustration of delta vectors and the displaced mesh.

The first step towards implementing this basic idea is to find a way to detect collisions between elements of a moving mesh. This is the subject of the following Section.

4.2 Continuous Collision Detection

Collision Detection (CD) is a classical problem within the fields of physical simulation and computer graphics [9]. In this work, the focus is on Continuous Collision Detection (CCD) which is a general class of collision detection techniques that differ from conventional CD in that the latter examines collisions at fixed intervals, checking whether contact has occurred between objects on a frame-by-frame basis. CCD, on the other hand, scans collisions over the entire trajectory of the object, taking into account the velocity, and thus prevents collisions from occurring between frames. CCD is a vast subject, with a rich literature – we refer the interested reader to the work of Wang et al. [25], that provides a detailed overview of the field with large-scale benchmarks of the various algorithms, and establishes the current state-of-the-art.

Assume that the mesh is composed of triangles and that the points move along a linear path in time. The possibility of a collision arises in two main cases: when a point collides with a triangle (vertex-face collision), or when an edge meets another edge (edge-edge collision). In both cases, the goal is to determine whether a contact occurred between time t_0 and t_1 , and if so, exactly when this contact occurred.

In the case of vertex-face collision, contact is investigated between a point p and three vertices of a triangle (v_1, v_2, v_3) . The CCD algorithm is used to calculate whether the point p touches the triangle within the time interval $[t_0, t_1]$. The contact occurs if the point falls in the plane of the triangle and is within the boundary of the triangle. In the case of edge-edge collision, we track the position in time of two edges, $e_1(p_1, p_2)$ and $e_2(p_3, p_4)$. The algorithm here examines whether the two edges intersect within $[t_0, t_1]$. This calculation is performed considering the minimum distance between edge pairs, and if the distance is zero, a collision occurs. In both cases the Time of Impact (TOI) is to be determined. The objective is to find the time (t) in a given interval (e.g. $t \in [0, 1]$ $t \in [0, 1]$) when the trajectories of two objects intersect.

One class of method for calculating the TOI reduce the problem to finding the roots of polynomials (i.e. where the distance between elements vanishes or falls below a tolerance threshold).

A direct way of solving this problem is to parameterize the trajectories with a parameter $t \in [0, 1]$ (i.e., $p_i(t) = (1-t)p_i^0 + tp_i^1$ and $v_i(t) = (1-t)v_i^0 + tv_i^1$) and formulate a *multivariate* polynomial whose roots correspond to intersections [6]. That is vertex-face CCD reduces to finding the roots of

$$F_{vf} : \Omega_{vf} = [0, 1] \times \{u, v \geq 0 | u + v \leq 1\} \rightarrow \mathbb{R}^3 \quad (4.1)$$

$$F_{vf}(t, u, v) = p(t) - ((1-u-v)v_1(t) + uv_2(t) + vv_3(t)). \quad (4.2)$$

Similarly for the edge-edge case the goal is to find the roots of

$$F_{ee} : \Omega_{ee} = [0, 1] \times [0, 1]^2 \rightarrow \mathbb{R}^3 \quad (4.3)$$

$$F_{ee}(t, u, v) = ((1-u)p_1(t) + up_2(t)) - ((1-v)p_3(t) + vp_4(t)) \quad (4.4)$$

In other words, the CCD problem reduces to determining if F has a root in Ω (i.e., there is a combination of valid t, u, v for which the vector between the point and the triangle is zero. Another class of methods are based on root finding for cubic *univariate* polynomials proportional to the volume of the tetrahedron spanned by the four points

in question, which vanishes when the points become co-planar [21]. While in theory any kind of multivariate or univariate root-finding method could be used to implement CCD, developing a procedure that is both efficient and remains robust when implemented in floating point arithmetic (in particular, does not produce false negatives, i.e. collision events that are not caught due to numerical errors) is very challenging. Indeed, in the large-scale benchmarks of [25] all previous methods produced some false negatives, despite theoretical guarantees that only hold when working with exact arithmetic. Some methods that performed better in this regard were on the other hand very inefficient.

To arrive at a CCD method that is maximally robust, while remaining as efficient as possible, [25] introduced an improved version of a classical interval-based method due to Snyder [22]. For more details, we refer to [25] and the references therein. In our experiments we used the open-source implementation of this algorithm, provided by the original authors¹.

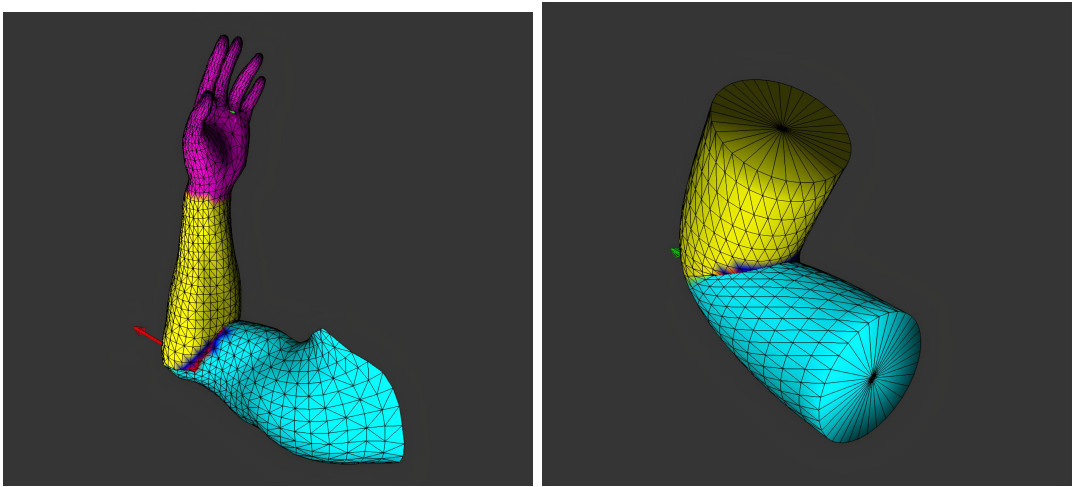


Figure 4.2: CCD for a Delta Mush deformation using the method of [25],

Figure 4.2 shows the result of CCD on a Delta Mush deformation, with vertex-face collisions shown in red, and edge-to-edge collisions shown in blue.

4.3 Delta Mush with CCD

Lets assume we run CCD for our Delta Mush deformation. To prevent self-intersection, a natural idea is to rescale the delta vectors of colliding vertices in proportion to their TOI. A naive approach would be to do a single sweep of CCD queries for each element based on the movement from the fully mushed mesh, and rescale the delta vectors based on the TOI of their vertices. This however would not be entirely correct, as stopping the movement of a set of vertices will also modify the trajectory of adjacent triangles and edges.

Instead we apply a more complex iterative approach. At every iteration we run CCD and identify the smallest TOI among all triangle-vertex and edge-edge collision events. Based on the minimal TOI, we scale each delta vector accordingly, and freeze the colliding elements in place for all subsequent iterations. These iterations are repeated until no colliding elements are found by CCD.

¹<https://github.com/Continuous-Collision-Detection/Tight-Inclusion>

In more detail, we make the assumption that the CCD algorithm returns a TOI between 0 and 1. Assume furthermore that we are currently in the i th iteration of CCD, and the last collision event happened at time t_{i-1} (we set $t_0 = 0$). The minimal TOI found by CCD is denoted τ and $V_i = \{v_1, \dots, v_4\}$ is the set of vertices involved in the corresponding collision. We set $t_i = t_{i-1} + \tau(1 - t_{i-1})$, scale by t_i all delta vectors of vertices not previously frozen, and mark vertices in V_i as frozen for subsequent iterations.

The steps of this algorithm are illustrated in Figure 4.3. The first picture shows the results of iteration 1, the second of iteration 5, the third of iteration 10 and the fourth of iteration 20. As can be seen, the deltas are adjusted to find the smallest TOI at the collision. Red indicates edge-to-edge collisions and blue indicates vertex-to-face collisions. The dots represent the actual collision points in the iteration step. It can be seen that using this method it is indeed possible to scale delta vectors so that self-intersections do not occur.

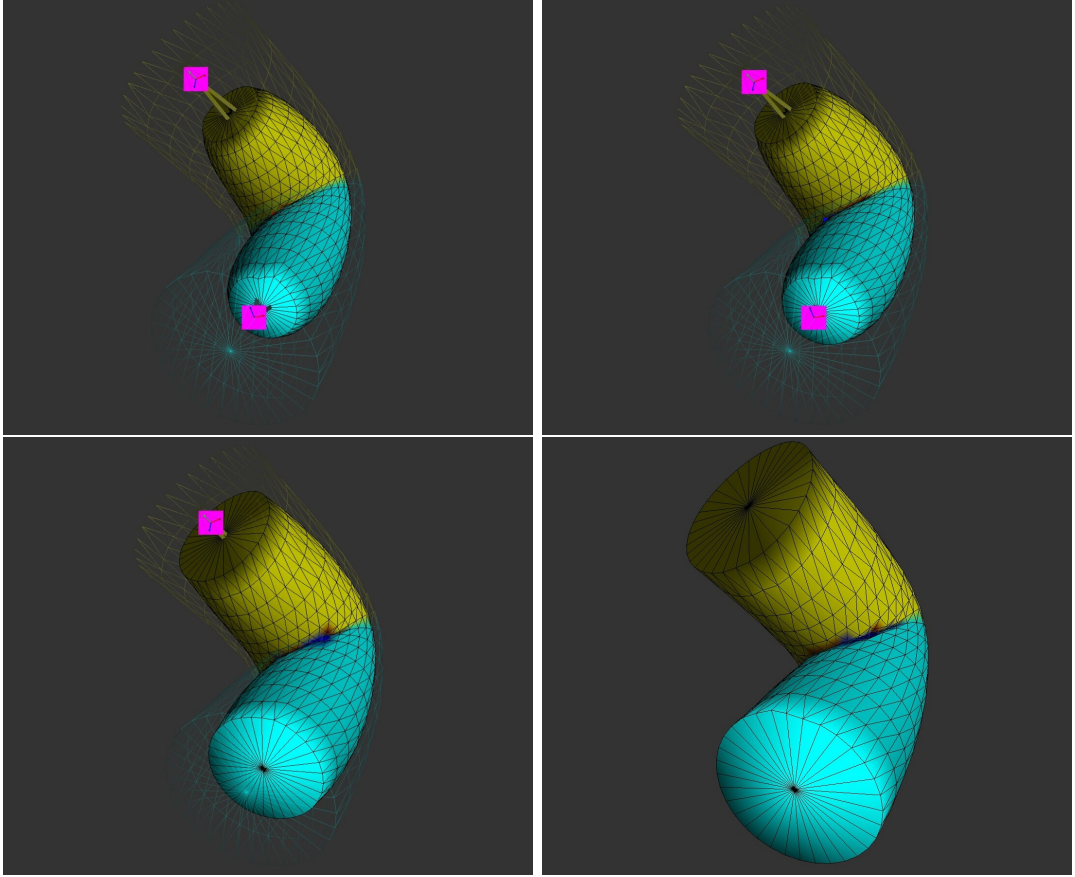


Figure 4.3: Iterations of our CCD-based Delta Mush algorithm.

4.4 Improving mesh quality

Our method as described might be capable of preventing self-intersections, but the shape quality of the resulting mesh can be very poor as the vertices might stop and. To improve the mesh quality we apply additional smoothing to its vertices during our iterations. We have already discussed ordinary Laplacian smoothing in Chapter 3, which can be thought of smoothing the surface in its normal direction. We also apply *tangential* smoothing, where the mesh vertices are constrained to move within the tangent plane of the surface. The process of tangential smoothing is carried out in several iterations, similar to Laplace smoothing, where a smooth surface is obtained by averaging the adjacent points. The main difference is that the neighbouring points are first projected onto the tangent plane of the central point before averaging is performed. This ensures that the smoothing does not result in shrinkage, as the points move only along the surface and not in any other direction. Through the iterative process, surface roughness can be subtly removed without changing the mesh too much. We compare the results with and without tangential smoothing on Figure 4.4.

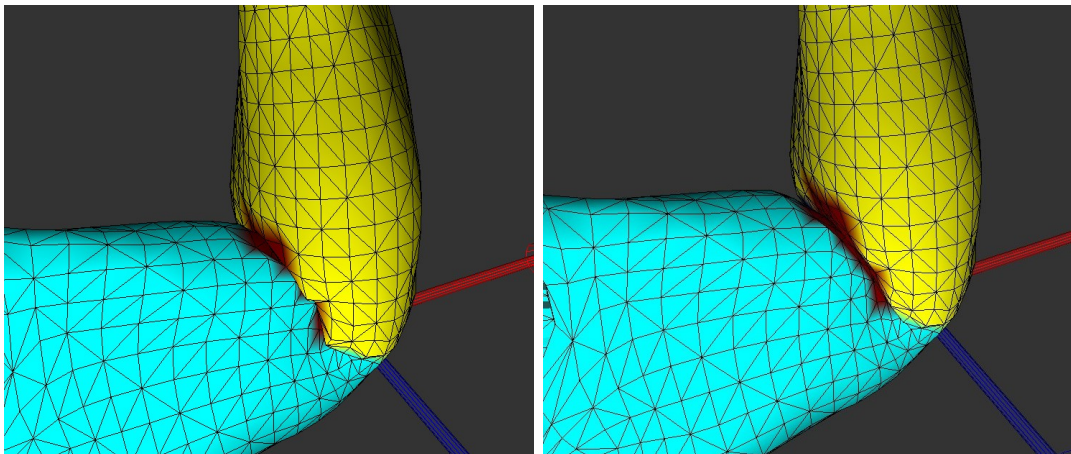


Figure 4.4: Illustration of tangential smoothing.

4.5 Improving performance

There are several approaches to optimizing the algorithm, since most collision checks involve examining vertices, tiles and edges that are unlikely to ever collide. It is therefore unnecessary to spend time and resources on these checks. One efficient solution is to inspect only certain relevant parts of the mesh, for example, a certain radius of the transformed node, since collisions are most likely to occur in the curved parts of the mesh. It is also in these locations that self-intersection is most likely to occur, so such local inspection is a way of increasing efficiency. We calculate the radius ratio depending on the size of the bone and the mesh around a connection point. In this radius, we mark the areas and run the CCD only on these areas, as this is where collisions are expected. In the example shown on Figure 4.5 we have highlighted in green the area selected for CCD checks.

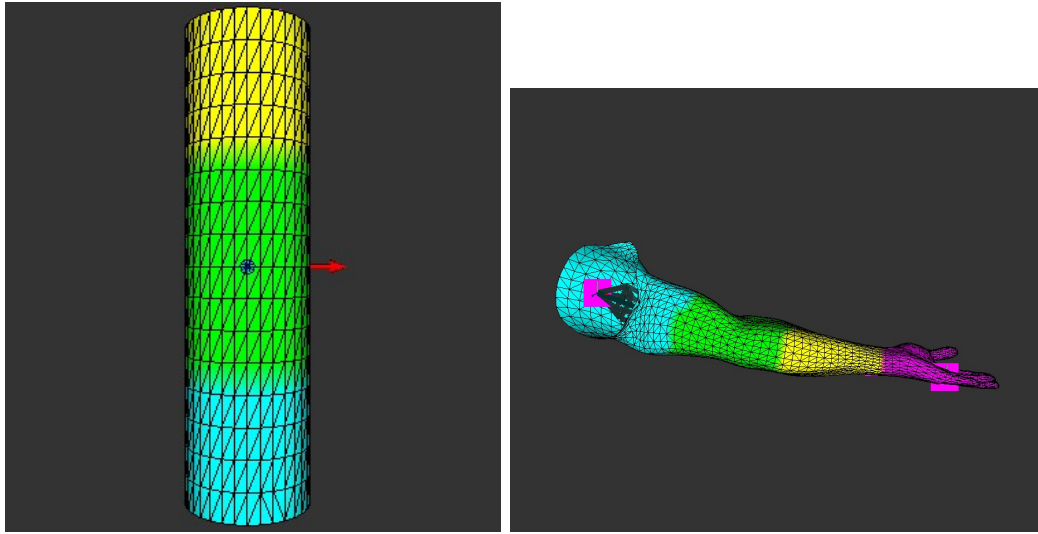


Figure 4.5: Illustration of area filtering for CCD.

A further optimization possibility is based on an Axis-Aligned Bounding Boxes (AABB), which effectively filters out vertices and edges that are certain not to collide. The AABB is a simple volume box that defines the minimum and maximum coordinates of the mesh or parts of the mesh along the x, y and z axes. The AABBs in question "enclose" pairs of mesh vertices/edges/triangles at the initial and final positions for the current CCD iteration. Given a pair of elements involved in a CCD query, it is quick and easy to check whether the two AABBs intersect. If two AABBs do not intersect, we can be sure that there will be no collision between these parts of the mesh, thus avoiding the need for a detailed CCD calculation. The AABB filtering provides a quick comparison option, as only the minimum and maximum coordinates need to be checked. This requires significantly fewer operations than detailed collision detection, so the overall performance of the algorithm can be improved. AABB greatly speeds up the application of our method for meshes that are large or have a more complex structure, as collision checks are limited to only potentially relevant regions.

Chapter 5

Results

In this Chapter we evaluate the effectiveness and performance of the proposed methods. All results in the report were produced using a prototype animation software developed by the author, that is based on a basic Qt¹ framework by P. Salvi², and uses the OpenMesh [4] and Eigen [12] libraries.

5.1 Deformation examples

We applied our method to a variety of example deformations, as shown in Figure 5.1, Figure 5.3, Figure 5.3, Figure 5.4, Figure 5.5, Figure 5.6, Figure 5.7, Figure 5.8 and Figure 5.9. On each figure, the result of the default Delta Mush deformation is shown on the left, while the result using the proposed CCD-based scaling of delta vectors is shown on the right. In each case self-intersections were resolved successfully. From the author’s personal experience as an animator, corrections such as that shown in Figure 5.4 would take hours manual work using the tools available in e.g. Blender.

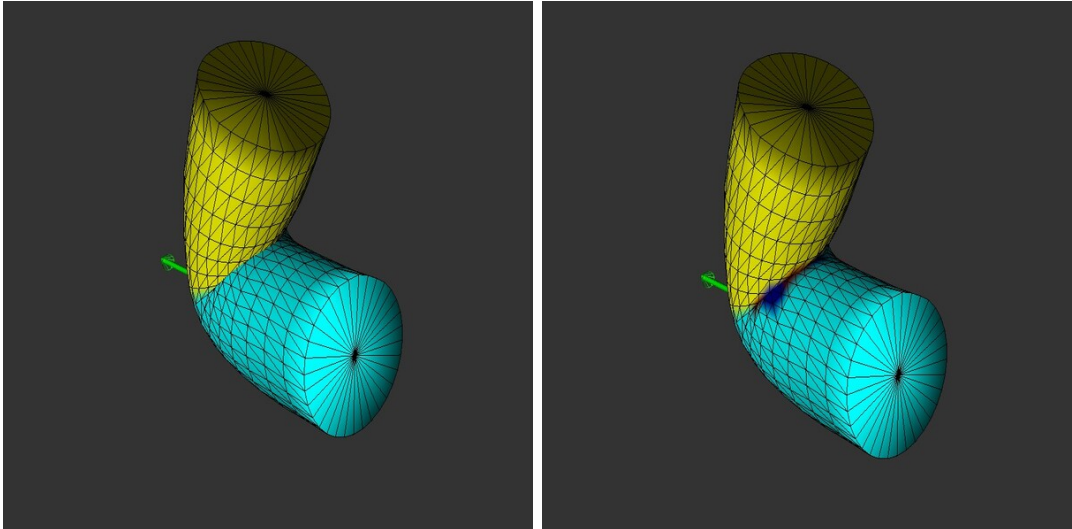


Figure 5.1: Deformation example 1.

¹<https://doc.qt.io/qt-5/>

²<https://github.com/salvipeter/sample-framework>

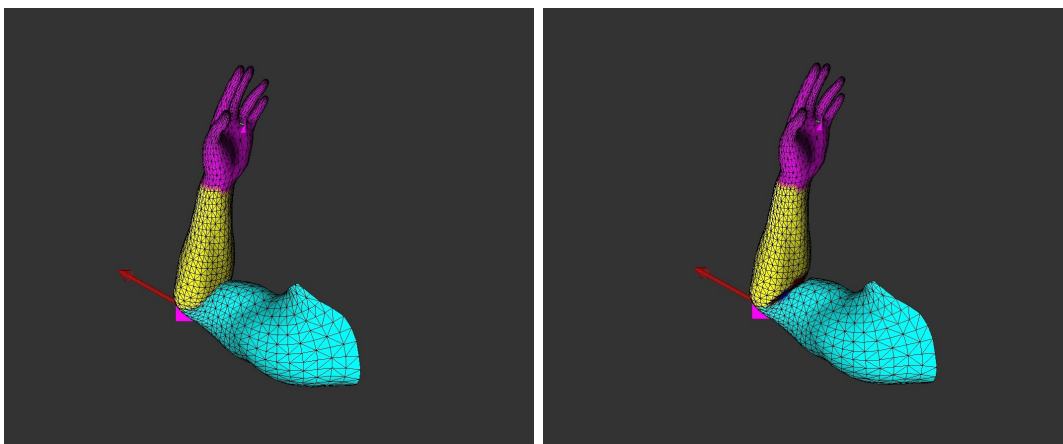


Figure 5.2: Deformation example 2.

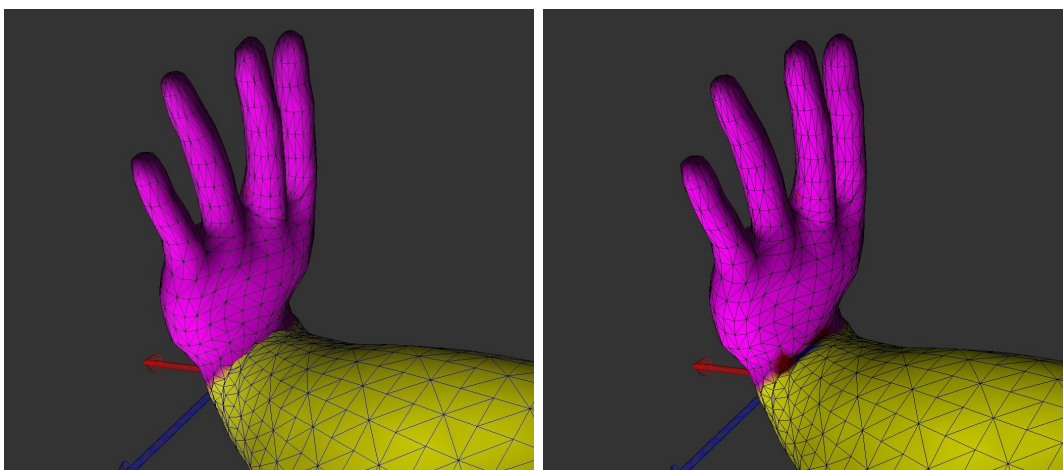


Figure 5.3: Deformation example 3.

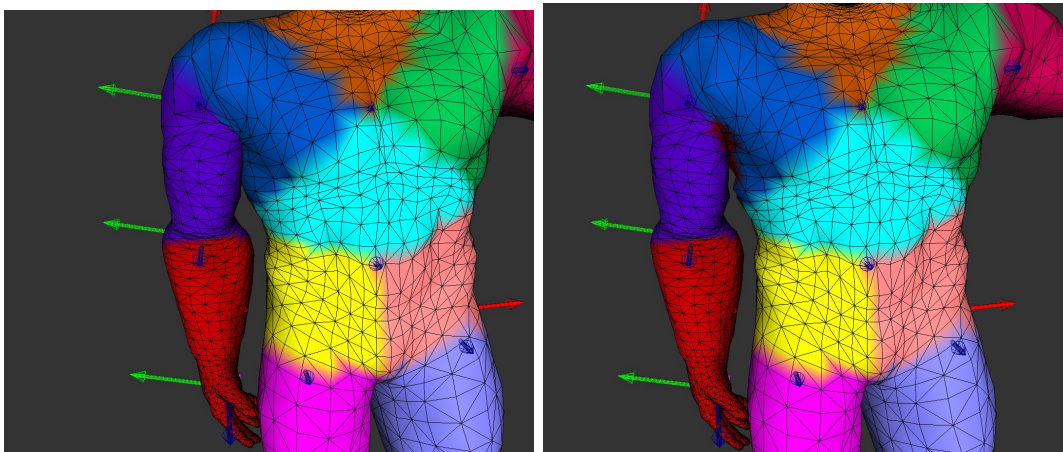


Figure 5.4: Deformation example 4.

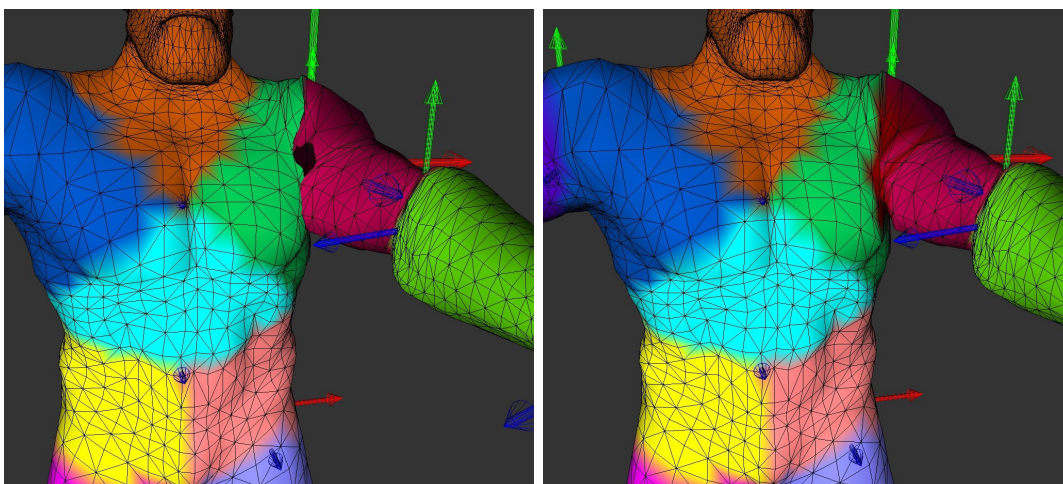


Figure 5.5: Deformation example 5.

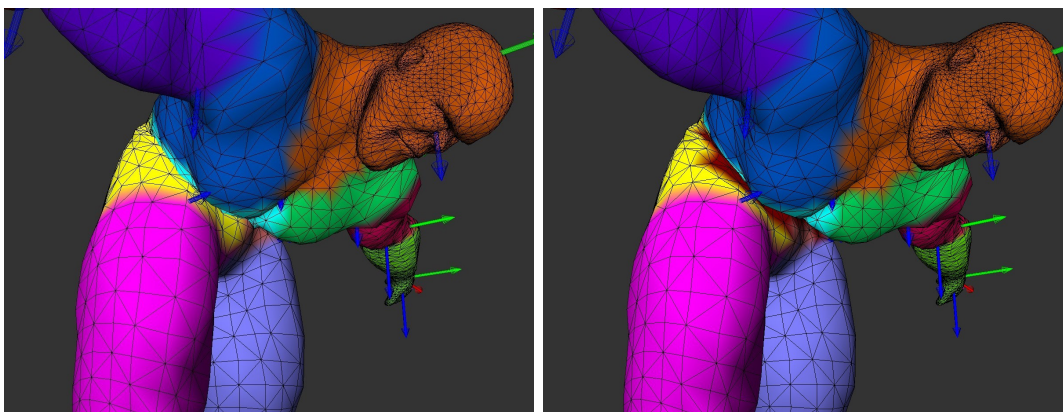


Figure 5.6: Deformation example 6.

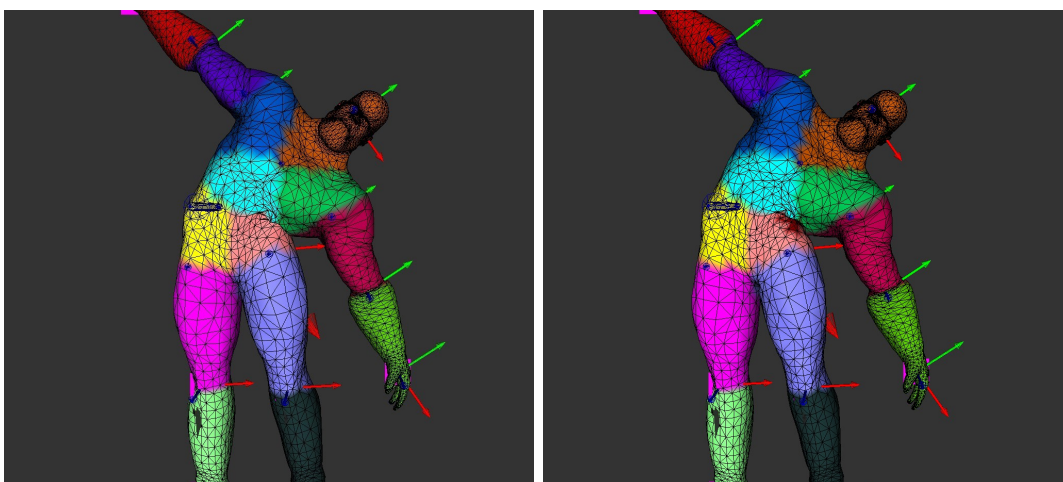


Figure 5.7: Deformation example 7.

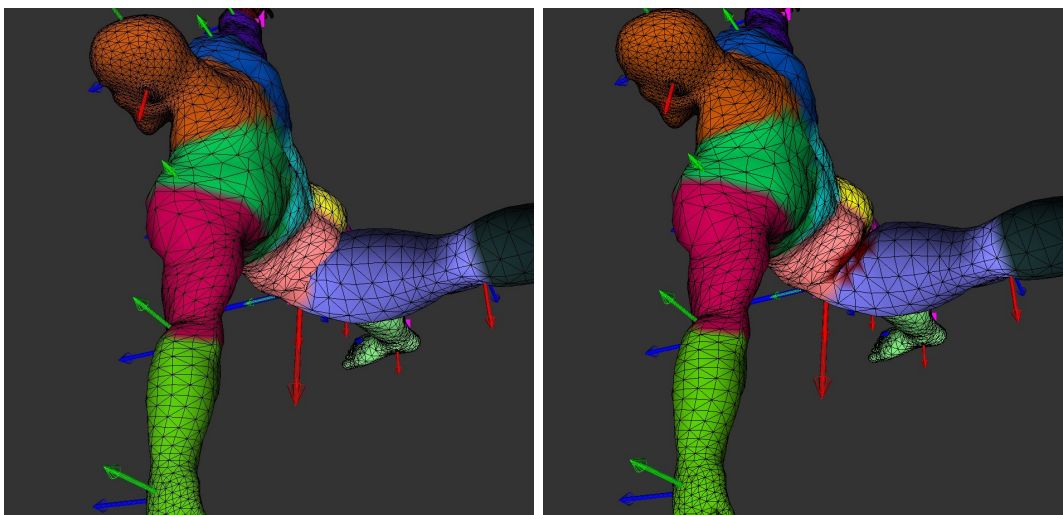


Figure 5.8: Deformation example 8.

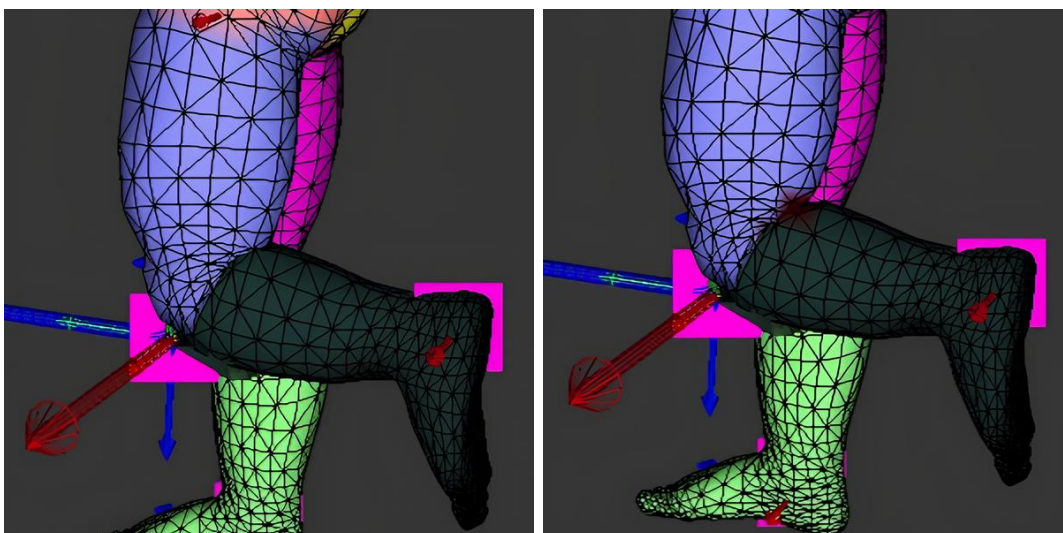


Figure 5.9: Deformation example 9.

5.2 Performance

Table 5.1 shows the data for different example meshes on which we tested the performance of our algorithm. These data illustrate how the algorithm performs for different geometries and complexities, and compares different meshes in terms of computational cost and collision detection accuracy.

name	vertex number	face number	edge number
Thanos	48996	16332	24498
Arm	12320	4104	6210
Cylinder	3008	1004	1632

Table 5.1: result of the models.

We compare the performance of our implementation (with and without the without the AABB-based filtering discussed in Chapter 3) on Figure 5.10 and Figure 5.11. As can be seen, without the filtering the method takes considerable time, but with filtering enabled the method runs much faster.

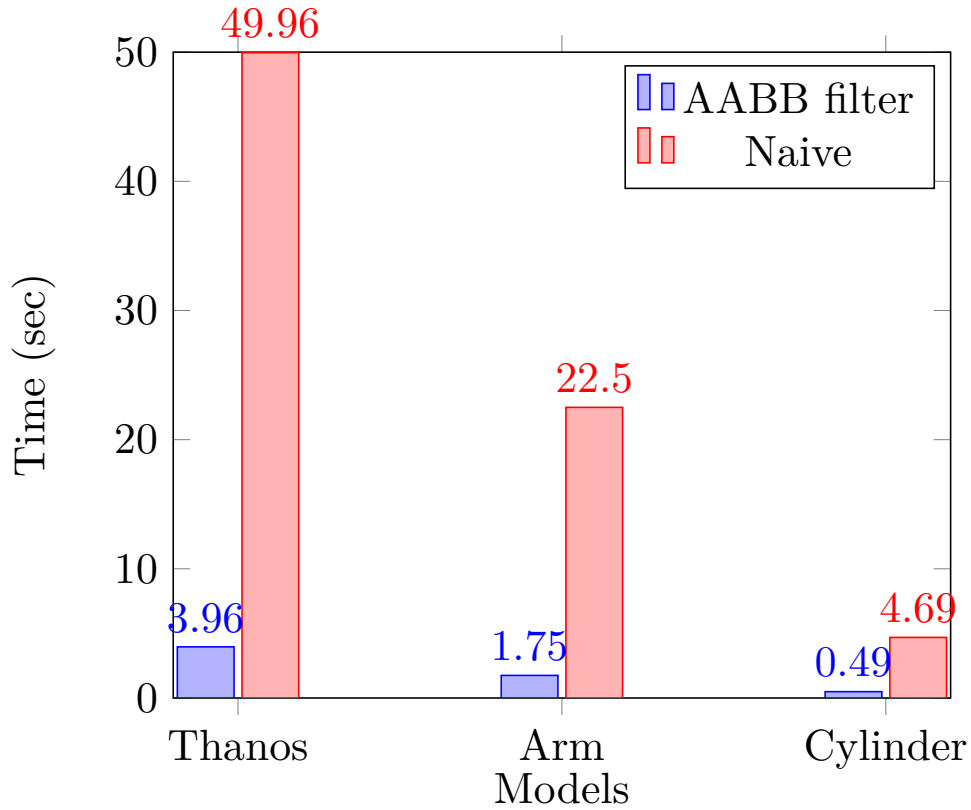


Figure 5.10: Runtime of CCD variants for different models.

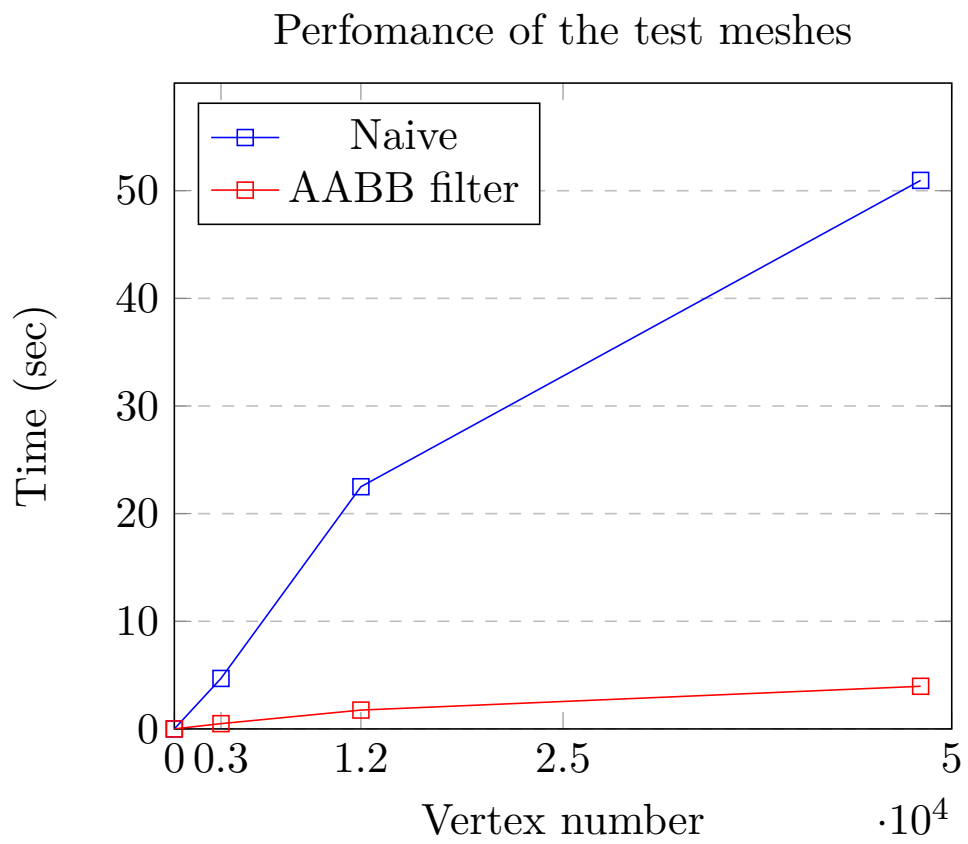


Figure 5.11: Performance scaling of CCD variants.

Chapter 6

Conclusion

6.1 Summary

In this work, we presented an improved method of delta mushing, specifically targeting the limitations of traditional techniques for dealing with self-intersections in character animation. By integrating continuous collision detection (CCD) and delta mush, we are able to accurately and efficiently detect and handle collisions, ensuring that the mesh geometry is not corrupted and no errors are introduced during delta mush based animation. Our approach has the potential to reduce the need for manual interventions by the user. Tests have shown that our technique performs well in a variety of deformation scenarios.

6.2 Future work

My future goal is to further optimize the method presented here, in particular the integration of continuous collision detection (CCD) and delta mush. During the optimization, I will strive to reduce the computation time to make the technique more suitable for the needs of real-time applications, such as video games and interactive virtual reality applications. In particular, I plan to improve the implementation of delta mush, based on the "direct" approach of [16]. Enhancing our approach to produce more realistic deformation with some kind of physical simulation also looks promising.

Finally, a very intriguing possibility is the integration of delta mush based deformation with the use of blendshapes [17] or sculpts [7]. Note that blendshapes are often prescribed as displacements (delta vectors) with respect to some skinned pose, which resembles the basic idea of delta mush – the key difference being the mesh from which the displacement are defined (the smoothed mesh for delta versus the skinned mesh for blendshapes). This suggests the possibility of a unified framework, where the mush becomes a central animation primitive and various deformer are defined as modifications of its delta vectors, thus carrying even further the original idea of delta mush deformer first outlined in [19].

Acknowledgements

I would like to thank Dr. Márton Vaitkus, my supervisor, without whom this work would not have been possible. His expertise, patience and guidance were indispensable throughout the research and helped me to cope with the challenges related to the further development of delta mush. He was always available and helpful when I got stuck during the work. I really enjoyed working with him and found every moment of his time valuable.

Thanks to my family and friends who supported me, encouraged me and assured me that they were always there for me. I ask for their understanding and patience so that I can devote all my attention to this work.

Bibliography

- [1] N. I. Badler and M. Morris. Modeling flexible articulated objects. In *Proc. Computer Graphics '82*, pages 305–314, 1982.
- [2] Ilya Baran and Jovan Popović. Automatic rigging and animation of 3d characters. *ACM Transactions on graphics (TOG)*, 26(3):72, 2007.
- [3] Mario Botsch and Olga Sorkine. On linear variational surface deformation methods. *IEEE transactions on visualization and computer graphics*, 14(1):213–230, 2007.
- [4] Mario Botsch, Stephan Steinberg, Stephan Bischoff, and Leif Kobbelt. OpenMesh - a generic and efficient polygon mesh data structure. In *1st OpenSG Symposium 2002*, 2002. <http://www.openmesh.org>.
- [5] Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Bruno Lévy. *Polygon Mesh Processing*. CRC Press, 2010.
- [6] Tyson Brochu, Essex Edwards, and Robert Bridson. Efficient geometrically exact continuous collision detection. *ACM Transactions on Graphics (TOG)*, 31(4):1–7, 2012.
- [7] Fernando de Goes, Patrick Coleman, Michael Comet, and Alonso Martinez. Sculpt processing for character rigging. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Talks*, pages 1–2, 2020.
- [8] Mathieu Desbrun, Mark Meyer, Peter Schröder, and Alan H Barr. Implicit fairing of irregular meshes using diffusion and curvature flow. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 317–324, 1999.
- [9] Christer Ericson. *Real-time collision detection*. CRC Press, 2004.
- [10] J. P. Lewis et al. Pose space deformation: A unified approach to shape interpolation and skeleton-driven deformation. In *SIGGRAPH*, pages 165–172, 2000.
- [11] N. Magnenat-Thalmann et al. Joint-dependent local deformations for hand animation and object grasping. In *Graphics Interface*, pages 26–33, 1988.
- [12] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. <http://eigen.tuxfamily.org>, 2010.
- [13] A. Jacobson. Skinning: Real-time shape deformation - automatic skinning via constrained energy optimization. In *SIGGRAPH Courses*, 2014.
- [14] L. Kavan. Skinning: Real-time shape deformation - direct skinning methods and deformation primitives. In *SIGGRAPH Courses*, 2014.

- [15] L. Kavan, S. Collins, J. Žára, and C. O’Sullivan. Geometric skinning with approximate dual quaternion blending. *ACM Transactions on Graphics (TOG)*, 27(4):1–23, 2008.
- [16] B. H. Le and J.P. Lewis. Direct delta mush skinning and variants. *ACM Transactions on Graphics (TOG)*, 38(4):1–13, 2019.
- [17] John P Lewis, Ken Anjyo, Taehyun Rhee, Mengjie Zhang, Frederic H Pighin, and Zhigang Deng. Practice and theory of blendshape facial models. *Eurographics (State of the Art Reports)*, 1(8):2, 2014.
- [18] William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3d surface construction algorithm. *ACM SIGGRAPH Computer Graphics*, 21(4):163–169, 1987.
- [19] Joe Mancewicz, Matt L. Derksen, Hans Rijpkema, and Cyrus A. Wilson. Delta mush: Smoothing deformations while preserving detail. In *SIGGRAPH*, 2014.
- [20] R. Parent. *Computer animation: algorithms and techniques*. Morgan Kaufmann, 3rd edition, 2012.
- [21] Xavier Provot. Collision and self-collision handling in cloth model dedicated to design garments. In *Computer Animation and Simulation’97: Proceedings of the Eurographics Workshop in Budapest, Hungary, September 2–3, 1997*, pages 177–189. Springer, 1997.
- [22] John M Snyder. Interval analysis for computer graphics. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 121–130, 1992.
- [23] Rodolphe Vaillant, Loïc Barthe, Gaël Guennebaud, Marie-Paule Cani, Damien Rohmer, Brian Wyvill, Olivier Gourmel, and Mathias Paulin. Implicit skinning: Real-time skin deformation with contact modeling. *ACM Transactions on Graphics (TOG)*, 32(4):1–12, 2013.
- [24] Rodolphe Vaillant, Gäel Guennebaud, Loïc Barthe, Brian Wyvill, and Marie-Paule Cani. Robust iso-surface tracking for interactive character skinning. *ACM Transactions on Graphics (TOG)*, 33(6):1–11, 2014.
- [25] Bolun Wang, Zachary Ferguson, Teseo Schneider, Xin Jiang, Marco Attene, and Daniele Panozzo. A large-scale benchmark and an inclusion-based algorithm for continuous collision detection. *ACM Transactions on Graphics (TOG)*, 40(5):1–16, 2021.