Improving Delta Mush Based Character Animation

Gergő Viktor Haragos, Márton Vaitkus

Abstract

Character animation is a classic problem in computer graphics. Traditionally, characters are modeled as polygonal meshes and animated using a skeleton. The standard method, linear blend skinning, transfers bone transformations to the skin through weighted averages [8]. While some tools automate skinning weight generation, manual fine-tuning by artists is still common. An alternative is the "Delta Mush" method [12], which involves: 1) animating with simple skinning weights, 2) smoothing the mesh to eliminate artifacts, and 3) restoring surface details by adding displacement vectors ("deltas"). Despite its simplicity and efficiency [9], Delta Mush does not prevent self-intersections. This work explores addressing this limitation by detecting self-intersections using continuous collision detection and modifying delta vectors for valid results.

Keywords: animation, delta mush, continuous collision detection, skinnning, self-intersection

1 Introduction

The 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 [13]. 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 [7], 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) [12]. 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).

Although DM alleviates some of the difficulties associated with the traditional animation workflow and can even be made suitable for real-time computation [9], it also has some well-known limitations. In particular, just like most other skinning methods, DM does not prevent the self-intersection of the mesh. The starting point of our 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 the current work is a method for resolving self-intersections during DM-based animations using the methods of Continuous Collision Detection (CCD). In section 2, an overview of the relevant technical background and related work is given, followed by a detailed description of Delta Mush in section 3. Then, in section 4, we present our proposed method for resolving self-intersections and present some practical results in section 5. Finally, the report concludes with an overview of various avenues for future research.

2 Previous Work

2.1 Skeletal Animation

In 3D animation, characters are represented by textured triangular meshes, called skins [13], which can contain thousands or millions of triangles. Animating each vertex individually is impractical, so skeletal animation is used. This technique links a surface mesh to a skeleton, a hierarchical set of "bones" that define poses and keyframes.

The skeleton, often starting in a default T-pose, controls mesh deformation through transformations like position, rotation, and scaling. Bones are typically arranged in a tree-like hierarchy, where moving one bone affects connected ones (e.g., moving a thigh moves the lower leg). The process of linking the skeleton's movement to the mesh is called skinning.

2.2 Skinning

Skinning is one of the most important techniques in 3D animation [8], 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, 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. 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.

2.3 Finding skinning weights

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 ,
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.

The Bone Heat method[1] models weight distribution as heat diffusion, ensuring smooth and natural results by solving an energy minimization problem. While effective, manual adjustments are often needed for complex models. Proper weight distribution is critical for realistic animations, as poorly defined weights lead to artifacts like stretching or collapsing. Advanced methods aim to reduce manual effort while maintaining quality. One alternative to automatic skinning is the Delta Mush method [12, 9], which forms the foundation of our current work.

2.4 Handling self-intersections, implicit skinning

Self-intersections occur when parts of a deformed mesh penetrate each other, leading to visually unacceptable results. Correcting these artifacts is essential, especially in high-quality animations. One approach involves manually creating "corrective blendshapes,"[10] where animators sculpt displacement vectors to fix intersections. While effective, this process is time-consuming and labor-intensive.

Implicit skinning [15] offers an automatic solution by representing the mesh as the level set of an implicit function. This guarantees collision-free deformations by blending Euclidean distance fields of submeshes corresponding to each bone. The final surface is extracted using methods such as marching cubes [11], or by displacing vertices to the implicit surface. Although robust, implicit skinning is computationally expensive and hardly suitable for real-time applications.

3 Delta Mush

Delta Mush is a method first proposed by Mancewicz et al. [12] and later improved and analyzed by Le and Lewis [9], 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 [7]. 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 – see Figure 1. Thus, by applying sufficiently strong smoothing to the deformed mesh, these artifacts can be made to disappear entirely.

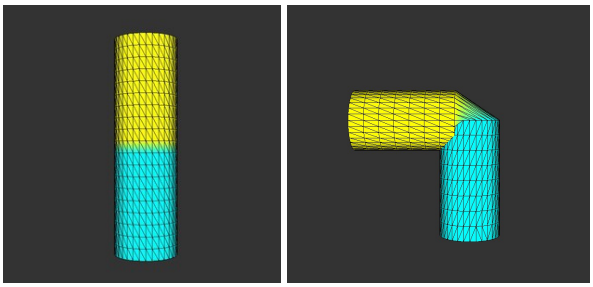

Figure 1: Rest pose (left) and LBS deformation with simple 0-1 weighting (right).

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 is

expected to be free of common skinning artifacts, it will also 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 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.

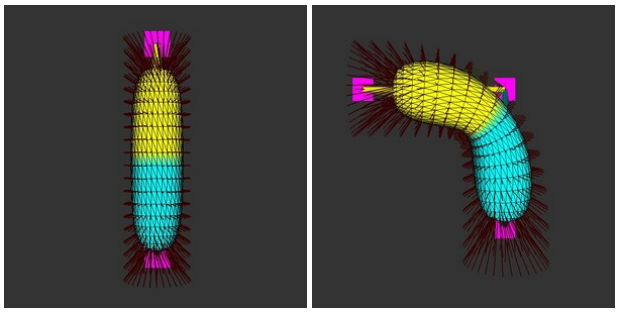

Figure 2: Illustration of the smoothed mesh and its delta vectors in the rest pose (left) and the deformed pose (right).

3.1 Laplacian smoothing

A crucial element of Delta Mush is the smoothing of the mesh. For this purpose, usually Laplacian smoothing [2] 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 towards its neighbours:

$$\mathbf{p}_i' = \sum_{j=0}^{N} w_j \mathbf{p}_j, \tag{1}$$

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.

3.2 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 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.

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 to be changed in each step of the animation to the normal, tangent and bi-tangent vectors of the current deformed (and smoothed) mesh, before the deltas are restored.

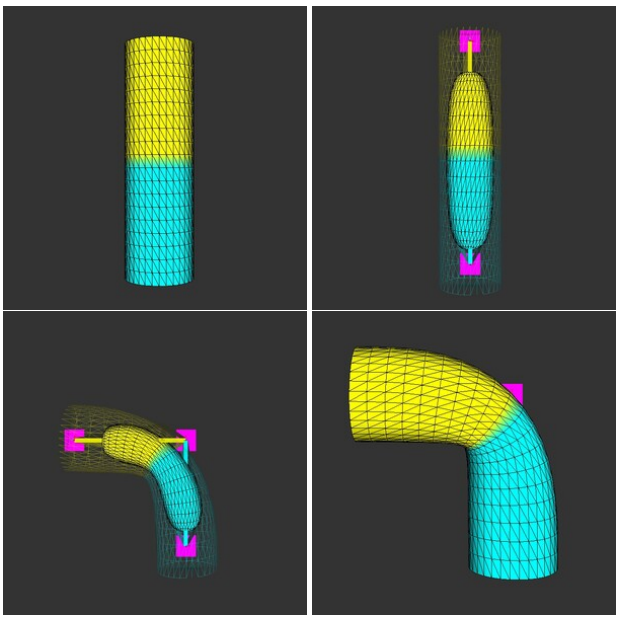

Figure 3: Illustration of the delta mush method: original rest pose mesh (top left), smoothed rest pose mesh (top right), smoothed deformed mesh (bottom left), and final deformation (bottom right).

4 Proposed Method

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.

4.1 Continuous Collision Detection

Collision Detection (CD) is a classical problem within the fields of physical simulation and computer graphics [5]. 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. [16], 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. $t \in [0,1]t\ [0,1]t \in [0,1]$) when the trajectories of two objects intersect.

To arrive at a CCD method that is maximally robust, while remaining as efficient as possible, [16] introduced an improved version of a classical interval-based method due to Snyder [14]. For more details, we refer to [16] and the references therein. In our experiments we used the open-source implementation of this algorithm, provided by the original authors[1].

4.2 Delta Mush with CCD

Let's 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.

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 ith 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 $V_i = \{v_1, \ldots, v_4\}$ 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. As can be seen, the deltas are adjusted to find the smallest TOI at the collision. Red color indicates edge-to-edge collisions and blue color 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.

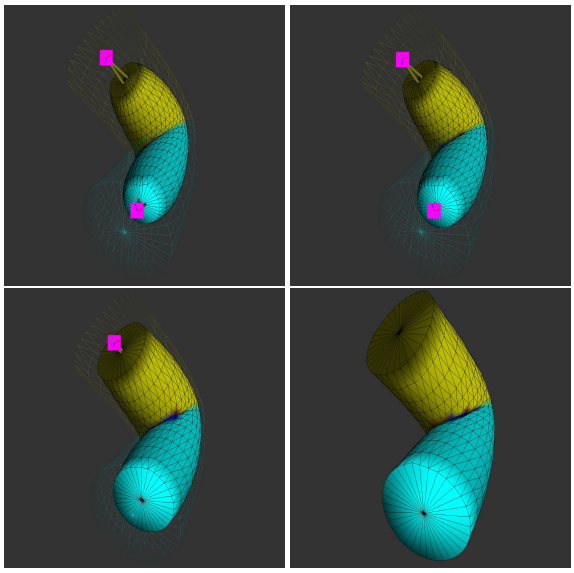

Figure 4: Iterations of our CCD-based Delta Mush algorithm (from left to right, and top to bottom: steps 1, 5, 10 and 20).

4.3 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. To improve the mesh quality we apply additional smoothing to its vertices during our iterations. We have already discussed ordinary Laplacian smoothing in section 3, which can be thought of smoothing the surface in its normal direction. We also apply *tangential* smoothing, where the mesh vertices are

[1] https://github.com/Continuous-Collision-Detection/Tight-Inclusion

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 5.

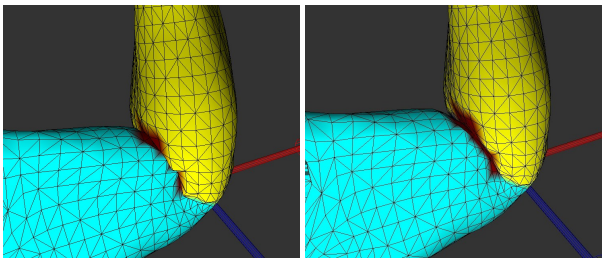

Figure 5: Illustration of tangential smoothing (left: without smoothing, right: with smoothing).

4.4 Improving performance

To optimize the algorithm, we can focus on reducing collision checks for vertices, tiles, and edges unlikely to collide. Instead of checking the entire mesh, we inspect areas within a specific radius of the transformed node, as collisions and self-intersections are most likely in curved regions. The radius is determined by the bone size and surrounding mesh, and CCD is run only in these marked areas.

Another optimization involves Axis-Aligned Bounding Boxes (AABB), which filter out mesh parts guaranteed not to collide. AABBs define the minimum and maximum coordinates along the x, y, and z axes for elements at their initial and final positions. If two AABBs don't intersect, no collision can occur, avoiding unnecessary CCD calculations. This method is computationally efficient, requiring only simple coordinate comparisons. AABB filtering significantly improves performance, especially for large or complex meshes, by limiting collision checks to relevant regions.

We compare the performance of our implementation with and without the AABB on Figure 6. As can be seen, without the filtering the method takes considerable time, but with filtering enabled the method runs much faster.

5 Results

In this Chapter we evaluate the effectiveness and performance of the proposed methods. All results in the report

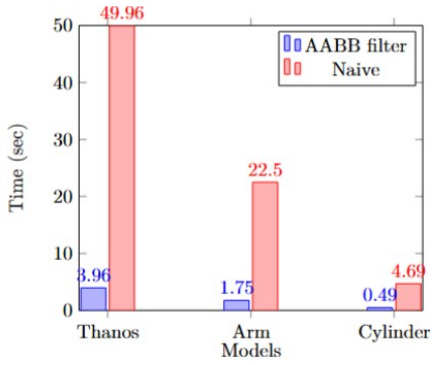

Figure 6: Runtime of CCD variants for different models.

were produced using a prototype animation software developed by the author, that is based on a basic Qt[2] framework by P. Salvi[3], and uses the OpenMesh [3] and Eigen [6] libraries.

We applied our method to a variety of example deformations, as shown in Figure 7, Figure 8, Figure 9, Figure 12, Figure 11, Figure 10. 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 our personal experience, corrections such as that shown in would take hours of manual work using the tools available in e.g. Blender.

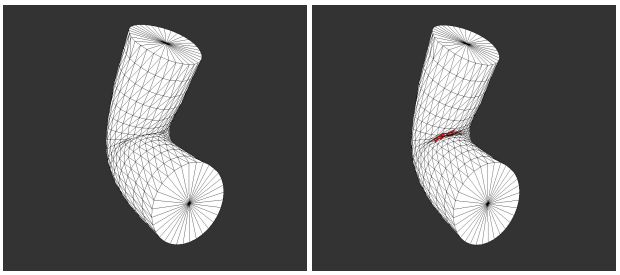

Figure 7: Example 1.

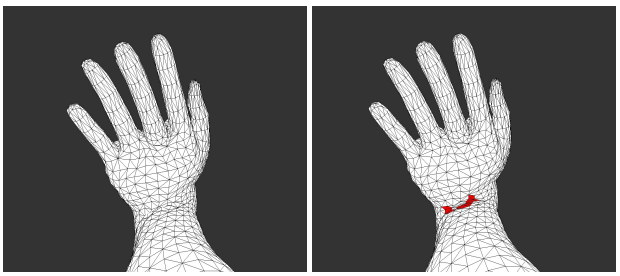

Figure 8: Example 2.

[2] https://doc.qt.io/qt-5/
[3] https://github.com/salvipeter/sample-framework

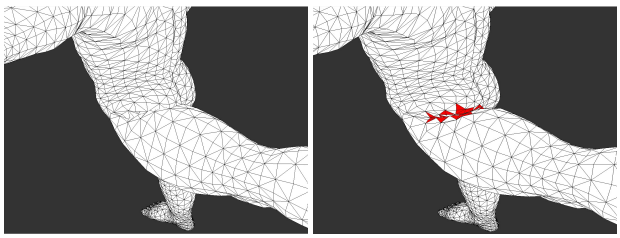

Figure 9: Example 3.

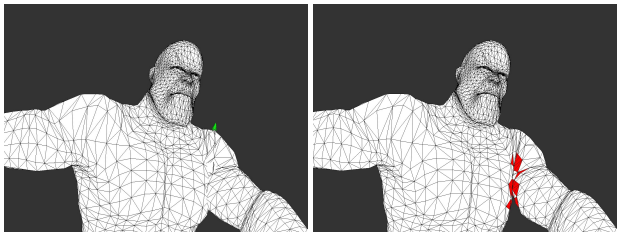

Figure 10: Example 4.

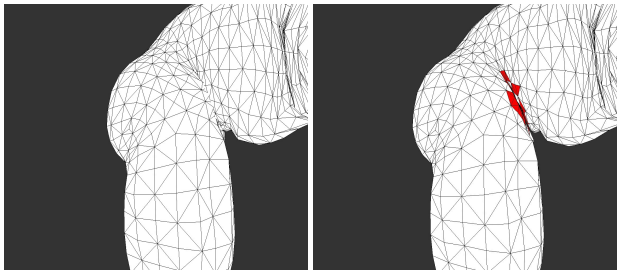

Figure 11: Example 5.

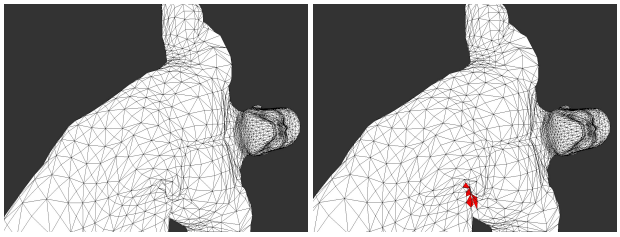

Figure 12: Example 6.

6 Conclusions and Future Work

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.

Our 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, we 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, we plan to improve the implementation of delta mush, based on the ”direct” approach of [9]. 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 [10] or sculpts [4]. Note that blendshapes are often prescribed as displacements (delta vectors) with respect to some skinned pose, which ressembles 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 deformers are defined as modifications of its delta vectors, thus carrying even further the original idea of delta mush deformers first outlined in [12].

Acknowledgements

This project has been supported by the Hungarian Scientific Research Fund (OTKA, No. 145970).

References

[1] Ilya Baran and Jovan Popović. Automatic rigging and animation of 3d characters. *ACM Transactions on graphics (TOG)*, 26(3):72, 2007.

[2] Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Bruno Lévy. *Polygon Mesh Processing*. CRC Press, 2010.

[3] 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`.

[4] 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.

[5] Christer Ericson. *Real-time collision detection*. CRC Press, 2004.

[6] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. `http://eigen.tuxfamily.org`, 2010.

[7] A. Jacobson. Skinning: Real-time shape deformation - automatic skinning via constrained energy optimization. In *SIGGRAPH Courses*, 2014.

[8] L. Kavan. Skinning: Real-time shape deformation - direct skinning methods and deformation primitives. In *SIGGRAPH Courses*, 2014.

[9] B. H. Le and J.P. Lewis. Direct delta mush skinning and variants. *ACM Transactions on Graphics (TOG)*, 38(4):1–13, 2019.

[10] 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.

[11] 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.

[12] Joe Mancewicz, Matt L. Derksen, Hans Rijpkema, and Cyrus A. Wilson. Delta mush: Smoothing deformations while preserving detail. In *SIGGRAPH*, 2014.

[13] R. Parent. *Computer animation: algorithms and techniques*. Morgan Kaufmann, 3rd edition, 2012.

[14] 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.

[15] 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.

[16] 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.