Deform, Cut and Tear a Skinned Model Using Conformal Geometric Algebra

Manos Kamarianakis[1,2(✉)] and George Papagiannakis[1,2]

[1] University of Crete, Heraklion, Greece
m.kamarianakis@gmail.com,george.papagiannakis@gmail.com
[2] ORamaVR, Heraklion, Greece
http://www.oramavr.com

Abstract. In this work, we present a novel, integrated rigged character simulation framework in Conformal Geometric Algebra (CGA) that supports, for the first time, real-time cuts and tears, before and/or after the animation, while maintaining deformation topology. The purpose of using CGA is to lift several restrictions posed by current state-of-the-art character animation & deformation methods. Previous implementations originally required weighted matrices to perform deformations, whereas, in the current state-of-the-art, dual-quaternions handle both rotations and translations, but cannot handle dilations. CGA is a suitable extension of dual-quaternion algebra that amends these two major previous shortcomings: the need to constantly transmute between matrices and dual-quaternions as well as the inability to properly dilate a model during animation. Our CGA algorithm also provides easy interpolation and application of all deformations in each intermediate steps, all within the same geometric framework. Furthermore we also present two novel algorithms that enable cutting and tearing of the input rigged, animated model, while the output model can be further re-deformed. These interactive, real-time cut and tear operations can enable a new suite of applications, especially under the scope of a medical surgical simulation.

Keywords: Conformal Geometric Algebra (CGA) · Skinning · Interpolation · Cutting algorithm · Tearing algorithm · Keyframe generation

1 Introduction

Skinned model animation has become an increasingly important research area of Computer Graphics, especially due to the huge technological advancements in the field of Virtual Reality and computer games. The original animation techniques, based on matrices [1] for translation, rotation and dilation, are still applied as the latest GPUs allow for fast parallel matrix operations. The fact that the interpolation result of two rotation matrices does not result in a rotation matrix, forced the use of quaternions as an intermediate step. The extra transmutation steps from matrix to quaternions and vice versa, adds some extra

performance burden to the animation but yields better results, solving problems such as the gimbal lock.

Nowadays, the state-of-the-art methods for skinned model animation use dual-quaternions, an algebraic extension of quaternions [13]. Dual quaternions handle both rotation and translation, while the dilation effect is still applied via matrices [12]. It is also noteworthy to mention that quaternions and dual quaternions enable blending techniques that resolve artifacts produced by simple linear blending, while further post-processing can be used to further minimize them [14].

Advances in Virtual Reality technology and the mass production of cheap VR headsets increased the demand of real-time simulation applications for both personal and industrial purposes. The research areas that sprout from these advancements, such as Virtual Surgery Simulation, require more complex model deformation such as cutting, tearing or drilling. Current algorithms [6,20] handle such deformations using tetrahedral mesh representation of the model, which demands a heavy pre-processing to be performed. Since originally introduced, cutting methods have been upgraded and polished to allow real-time results, using mostly finite element methods and clever optimization [4,7,15]. To make the final results even more realistic, physics engines utilizing position-based dynamics are used to simulate soft-tissue cuts at the expense of performance [2,3,5].

Our approach utilizes the Conformal Geometric Algebra (CGA) framework to perform both model animation and cutting. CGA is an algebraic extension of dual-quaternions, where all entities such as vertices, spheres, planes as well as rotations, translations and dilation are uniformly expressed as *multivectors* [8,11,19]. The usage of multivectors allows model animation without the need to constantly transmute between matrices and (dual) quaternions, enabling dilation to be properly applied with translation [17,18]. Furthermore, the interpolation of two multivectors of the same type correctly produce the expected intermediate result [9], which makes creation of keyframes trivial to implement. Finally, usage of the proposed framework demands a single representation type for all data and results, which is the current trend in computer graphics [16].

Our Contribution: The novelty of our work initially involves the complete implementation of rigged model animation in terms of CGA, extending the work of Papaefthymiou et al. [17] with full python-based implementation that enables keyframe generation on-the-fly. The original animation equation involving matrices is translated to its equivalent multivector form (see Sect. 3.1) and all information required to apply the formula (vertices, animation data) is obtained from the model and translated to multivector. This enables us to have future animation models in CGA representation only, which, in combination with an optimized GPU multivector implementation, would produce faster results under a single framework. A novelty of our work is the cutting and tearing algorithm that is being applied on top of the previous framework; given the input animated model, we perform real-time cuts and tears on the skin and then further re-deform the output model. The subpredicates used in these two algorithms

utilize the multivector form of their input, so they can be implemented in a CGA-only framework. Their design was made in such a way that little to no pre-processing of the input model is required while allowing a future combination with a physics engine. Furthermore, using our method, we can generate our own keyframes in real-time instead of just interpolating between pre-defined ones. Our all-in-one cpu python implementation is able to process an existing animation model (provided in .dae or .fbx format) and translate the existing animation in the desired CGA form while further tweaks or deformations are available in a simple way to perform. Such an implementation is optimal as far as rapid prototyping, teaching and future connection to deep learning is concerned. It also constitutes the base for interactive cutting and tearing presented in Sect. 3.2.

2   State of the Art

The current state of the art regarding skeletal model animation is based on the representation of bones animation via transformation matrices and quaternions or dual-quaternions. Such an implementation allows for efficient and robust interpolation methods between keyframes; linear interpolation of the quaternions is done in a naive and easy to perceive way. A major drawback of such an implementation is that a dilation method can not be applied as a scaling matrix always refers to the origin and not the parent bone [17].

To be more precise regarding the mechanics of the animation process, in the case of a simple animated model, every bone b_i amounts to an offset matrix O_i and an original transformation matrix t_i. The skin of the model is imported as a list of vertices v and a list of faces f. A bone hierarchy is also provided where $\{t_i\}$ are stored along with information regarding the animation of each joint. This information, usually referred to as *TRS data*, is provided in the form of a quaternion, a translation vector and a scaling vector that represent respectively the rotation, displacement and scaling of the joint with respect to the parent joint for each keyframe (see Sect. 2.1).

In order to determine the position of the skin vertices at any given time k and therefore render the scene by triangulating them using the faces list, we follow the steps described below. Initially, a matrix G is evaluated as the inverse of the transformation matrix that corresponds to the root node. Afterwards, we evaluate the *global transformation matrix* for every bone b_i at time k and denote it as $T_{i,k}$. To evaluate all $T_{i,k}$, we recursively evaluate the matrix product $T_{j,k}t_{i,k}$ where b_j is the parent bone of b_i, given that $T_{r,k}$ is the identity matrix (of size 4), where b_r denotes the root bone. The matrix $t_{i,k}$ is a transformation matrix equal to t_i if there is no animation in the model; in this case, our implementation allows to generate the keyframes ourselves in real-time. Otherwise, $t_{i,k}$ is evaluated as

$$t_{i,k} = TR_{i,k}MR_{i,k}S_{i,k} \tag{1}$$

where $TR_{i,k}, MR_{i,k}, S_{i,k}$ are the interpolated matrices that correspond to the translation, rotation and scaling of the bone b_i at a given time k.

After evaluating the matrices $\{T_{i,k}\}$ for all bones $\{b_i\}$, we can evaluate the global position of all vertices at time k, using the *animation equation*:

$$V_k[m] = \sum_{n \in I_m} w_{m,n} GT_{n,k} O_n v[m] \tag{2}$$

where

- $V_k[m]$ denotes the skin vertex of index m (in homogeneous coordinates) at the animation time k,
- I_m contains up to four indices of bones that affect the vertex $v[m]$,
- $w_{m,n}$ denotes the "weight", i.e., the amount of influence of the bone b_n on the vertex $v[m]$,
- O_n denotes the offset matrix corresponding to bone b_n, with respect to the root bone,
- G denotes the inverse of the transformation matrix that corresponds to the root bone (usually equals the identity matrix) and
- $T_{n,k}$ denotes the deformation of the bone b_n at animation time k, with respect to the root bone.

2.1   State-of-the-Art Representation

The modern way to represent the TRS data of a keyframe is to use matrices for the translation and dilation data as well as quaternions for the rotation data. Let $\{TR_i, R_i, S_i\}$, denote such data at keyframe $i \in \{1, 2\}$, where:

- $TR_i = \begin{bmatrix} 1 & 0 & 0 & x_i \\ 0 & 1 & 0 & y_i \\ 0 & 0 & 1 & z_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$ and $S_i = \begin{bmatrix} sx_i & 0 & 0 & 0 \\ 0 & sy_i & 0 & 0 \\ 0 & 0 & sz_i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ represent the translation by
  (x_i, y_i, z_i) and the scale by (sx_i, sy_i, sz_i) respectively and
- R_i is a quaternion representing the rotation.

Before quaternions, euler andgles and the derived rotation matrices were used to represent rotation data. However the usage of such matrices induced a great problem: a weighted average of such matrices does not correspond to a rotation matrix and therefore interpolating between two states would require interpolating the euler angles and re-generate the corresponding matrix. This in turn would sometimes lead to a gimbal lock or to 'candy-wrapper' artifacts such as the ones presented in [12].

The usage of quaternions allowed for easier interpolation techniques while eradicating such problems. Nevertheless, a transformation of the interpolated quaternion to corresponding rotation matrix was introduced since the GPU currently handles only matrix multiplications in a sufficient way for skinning reasons. Therefore, the interpolation between the two keyframes mentioned above follows the following pattern:

1. the matrices $TR_a = (1-a)TR_1 + aTR_2$ and $S_a = (1-a)S_1 + aS_2$ are evaluated for a given $a \in [0, 1]$,

2. the quaternion $R_a = (1 - a)R_1 + aR_2$ is determined and finally,
3. the rotation matrix MR_a that corresponds to R_a is calculated.

The interpolated data TR_a, MR_a and S_a are then imported to the GPU in order to determine the intermediate frame, based on the Eq. (2).

Using the method proposed in this paper, all data are represented in multivector form. A major implication of this change is that the interpolation between two states is done in a more clear and uniform way as presented in Sect. 3. This also makes the need to constantly transform a quaternion to a rotation matrix redundant, although we are now obliged to perform multivector additions and multiplications as well as down project points from $\mathbb{R}_{4,1}$ to $\mathbb{R}_3$ to parse them to the GPU. However, since all our data and intermediate results are in the same multivector form, we could (ideally) program the GPU to implement such operations and therefore greatly improve performance.
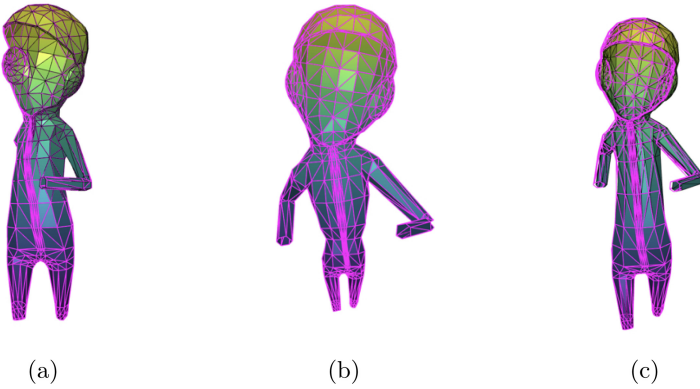

(a)   (b)   (c)

Fig. 1. Skinning via multivectors versus skinning via dual quaternions. The original model is deformed using multivectors and depicted in magenta wireframe, superimposed with the color-graded result (based on the z coordinate of each vertex) of the quaternion method for the same deformation. It is qualitatively verified that linear blending of multivectors produces similar results with the current state-of-the-art method. Evaluating the vector differences of all vertices for the two methods, we have evaluated the approximation error assuming the quaternion method to be the correct, using the infinity (ℓ_∞) norm. (a) Applying rotation on a bone, approximation error 0.3%. (b) Applying dilation on a bone, approximation error 0.00035%. (c) Applying translation, approximation error 1%. The model used contains 1261 vertices and 1118 faces.
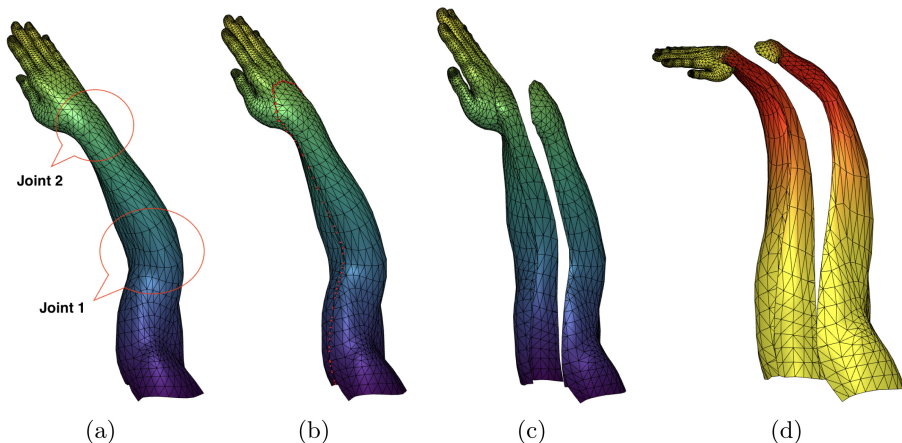
(a)   (b)   (c)   (d)

Fig. 2. Cutting module intermediate steps. (a) The original animated model. (b) The model where the (red) intersection points of the cutting plane and the mesh are calculated and re-triangulated. (c) The model after the cut. (d) The model is deformed by a rotation (axis = $(0, 1, 1)$, 0.7 rad), a translation (vector = $(13, 0, 0)$) and a dilation (factor = 0.5) at joint 1 (elbow), as well as another rotation (axis = $(0, 1, 1)$, 0.3 rad) at joint 2 (wrist). Note that minimal artifacts occur in the final result. The vertices in (d) are colored depending on the influence of joint 1 which is mostly deformed. The vertices in (a)–(c) are colored based on their z coordinate. (Color figure online)

3   Our Algorithms and Results

3.1   Multivector Form of the Animation Equation

The animation Eq. (2), core of the animation algorithm, yields fast results (especially when combined with a GPU implementation) but denies us a robust way to dilate with respect to a bone. Our motivation is to extend and apply the animation equation for multivector input as proposed in [17].

To be more specific regarding our method, we propose the replacement of all matrices appearing in (2) with multivectors for animation purposes. The transformation matrix of t_i of each bone b_i as well as all information regarding translation and rotation for each keyframe can be easily converted to multivectors [8,11]. Consequently, we can evaluate the multivector $M_{i,k}$ which is equivalent to the matrix $T_{i,k}$ by following the same procedure of determining the latter (described in Sect. 2) while substituting all involved matrices with the corresponding multivectors.

Note that various techniques can be used to interpolate between two keyframes to obtain $M_{i,k}$; for existing keyframes logarithmic blending is preferred [9,12], whereas for keyframe generation we use linear blending. In both scenarios, the intermediate results are multivectors of the correct type.

Furthermore, each offset matrix O_n and each skin vertex $v[m]$ is translated to their CGA form B_n and $c[m]$ respectively. Finally, G matrix is normalized to identity and is omitted in the final equation.

Our final task is to translate in CGA terms the matrix product $T_{n,k}O_n v[m]$, where apparently each multiplication sequentially applies a deformation to vertex $v[m]$. To apply the respective deformations, encapsulated by $M_{n,k}$ and B_n, to CGA vertex $c[m]$, we have to evaluate the *sandwich geometric product* $(M_{n,k}B_n)c[m](M_{n,k}B_n)^\star$ where $V^\star$ denotes the *inverse* multivector of V (see [11,13] for details).

Summarizing, if the multivector form of the vertex $V_k[m]$, which corresponds to the final position of the m-th vertex at animation time k, is denoted by $C_k[m]$, then the *multivector animation equation* becomes

$$C_k[m] = \sum_{n \in I_m} w_{m,n}(M_{n,k}B_n)c[m](M_{n,k}B_n)^\star \tag{3}$$

After the evaluation of $C_k[m]$ for all m, we can down-project all these conformal points to the respective euclidean ones in order to represent/visualize them and obtain the final result of the keyframe at time k.

The replacement of matrices with multivectors enables the introduction of dilations in a simple way. The multivector $M_{i,k}$ that represents a rotation and translation with respect to the parent bone of b_i can be replaced with $M_{i,k}D_{i,k}$ where $D_{i,k}$ is the corresponding dilator and the operation between them is the geometric product. The dilator corresponds to a scale factor with respect to the parent bone, information that could not be easily interpreted via matrices. However, since the application of a motor and/or a dilator to a vertex is a sandwich operation, such a dilation becomes possible when using multivectors.

A comparison between the results of our proposed method and the current state-of-the-art is shown in Fig. 1, where we successfully apply dilation to different bones and obtain similar results. Rotations, dilations and translations are obtained in our method using multivectors only, under a single framework with simpler notation/implementation; linear blending is used to interpolate between keyframes.

3.2   Cutting and Tearing Algorithms

A novelty we present in this paper is the cutting and tearing algorithms on skinned triangulated models. As the name suggests, the first module enables the user to make a planar cut of the model whereas the latter is used to perform smaller intersections on the skin. In the following sections, we provide a detailed presentation of the algorithms involved as well as certain implementation details.

Cutting Algorithm. Cutting a skinned model is implemented in current bibliography in many forms [6]. The most common technique is via the usage of tetrahedral meshes which require a heavy preprocessing on the model and currently do not enable further animation of the model. Our work includes an algorithm to planar cut a model (or a part of it) where the final mesh is deformable, as we implemented a function to calculate weights for all additional vertices that did not originally exist (see Fig. 2). Most of the subpredicates used in the cutting

algorithm are implemented in terms of conformal geometry and therefore can be used even if the model is provided in multivector form.

Our proposed planar cut implementation is summarized as Algorithm 1. A description of how we tackle the weight evaluation in step 4 is found in Sect. 3.3. Our algorithm does not require tetrahedral meshed models and requires minimum to none preprocessing. It is GA-ready and the low number of operations it demands make it suitable for VR implementations.
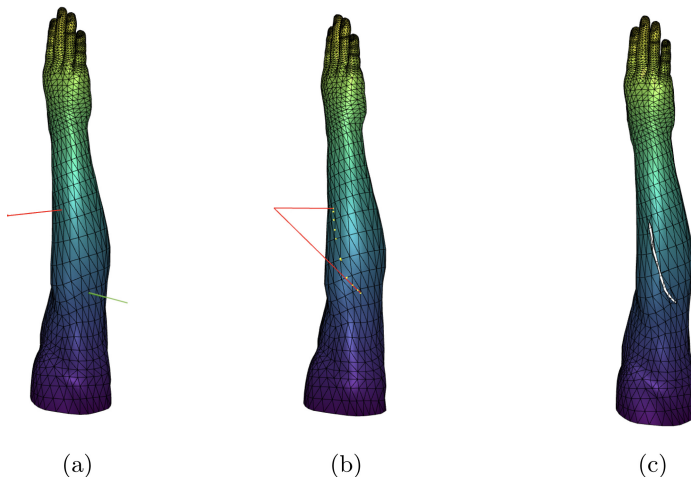

(a)   (b)   (c)

Fig. 3. Tearing module intermediate steps. (a) The original animated model and the scalpel's position at two consecutive time steps. (b) The plane defined by the scalpels (depicted as a red tringle) intersects the skin in the magenta points. (c) The intermediate points are used in the re-triangulation, and are «pushed»away from the cutting plane to form an open tear.

Tearing Algorithm. The purpose of this module is to enable partial cuts on the skinned model, in contrast with the cutting module where the cut is, in a sense, complete. The importance of this module derives from the fact that most

---

Algorithm 1. Cutting Algorithm

Input: Triangulated Mesh $M = (v, f)$ (f is the face list), and a plane Π.
Output: Two meshes $M_1 = (v_1, f_1)$ and $M_2 = (v_2, f_2)$, result of M getting cut by Π.
 1: Evaluate (using GA) and order the intersection points of Π with each face of M.
 2: Evaluate the weights and bone indices that influence these points.
 3: Re-triangulate the faces that are cut using the intersection points.
 4: Separate faces in f_1 and f_2, depending on which side of the plane they lie.
 5: From f_1 and f_2, construct M_1 and M_2.

of the surgical incisions are partial cuts and therefore they are worth replicating in the context of a virtual surgery. Towards that direction, our work involves an algorithm that both tears a skinned model and also enables animation of the final mesh (see Figs. 3 and 4).

To understand the philosophy behind the design of the tearing algorithm that is described below, one must comprehend the differences between cutting and tearing. In tearing, the movement of a scalpel defines the tear rather than a single plane. To capture such a tear in geometric terms, we have to take into consideration the location of the scalpel in either a continuous way (e.g. record the trail of both endpoints of the scalpel in terms of time) or a discrete way (e.g. know the position of the scalpel at certain times t_i). For VR purposes, the latter way is preferred as it yields results with better fps, since input is hard to be monitored and logged continuously in a naive way. For these reasons, our implementation requires the scalpel position to be known for certain t_i.

The proposed tearing algorithm is summarized in Algorithm 2. A description of how we tackle the weight evaluation in step 4 is found in Sect. 3.3.

Algorithm 2. Tearing Algorithm

Input: Triangulated Mesh $M = (v, f)$, and scalpel position at time steps t_i and t_{i+1}
Require: Scalpel properly intersects M at these time steps
Output: The mesh $M_t = (v_t, f_t)$ resulting from M getting torn by the scalpel
 1: Determine the intersection points S_i and S_{i+1} of M with the scalpel at time step t_i and t_{i+1} respectively.
 2: Determine the plane Π, containing S_i and the endpoints of scalpel at time t_{i+1}. Small time steps guarantee that Π is well-defined.
 3: Evaluate the intersection points Q_j of Π and M, s.t. the points $S_i,Q_0,Q_1,\ldots,Q_m$, S_{i+1} appear in this order on Π when traversing the skin from S_i to S_{i+1}.
 4: Assign weights to points S_i, S_{i+1} and all Q_j.
 5: Re-triangulate the torn mesh, duplicating Q_j vertices.
 6: Move the two copies of Q_j away from each other to create a visible tear (optional).

Our major assumption is that all intermediate intersection points lie on this plane, which is equivalent to the assume that the tearing curve is smooth, given that t_i and t_{i+1} are close enough. In our implementation, during step 6, the intermediate torn points are moved parallel to the direction of the normal of the plane Π and away from it, to replicate the opening of a cut human tissue.

3.3   Implementation Details, Performance and Video Results

The main framework used for skinning and animation with the use of multi-vectors is Python's PyAssimp[1] and Clifford[2] package for the evaluation of the vertices and the Meshplot package for rendering the model. The use of Python

---

[1] PyAssimp Homepage: https://pypi.org/project/pyassimp/.
[2] Clifford Homepage: https://clifford.readthedocs.io/.

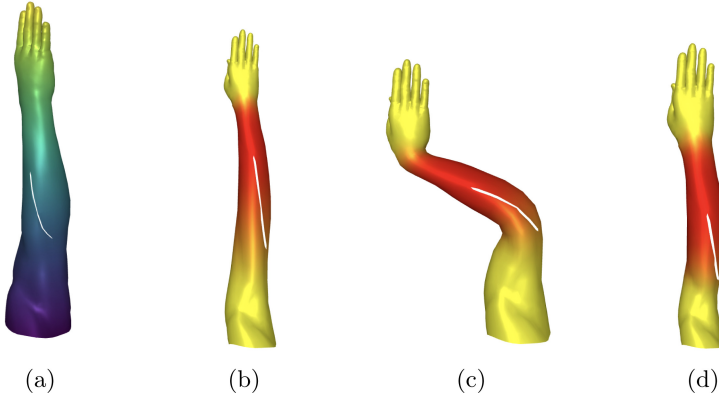|			
(a)	(b)	(c)	(d)

Fig. 4. Deformation of a torn model. (a) The original model after applying the tear. (b) Two rotations are applied to the torn model, one at elbow joint around y-axis by -1 rad, and another at wrist joint around y-axis by 1 rad. (c) A dilation of scale 1.5 is applied to the torn model, at elbow joint. (d) A translation is applied to the torn model at elbow joint with translation vector $(18, 0, 0)$. In all cases, minor artifacts only arise, despite the great magnitude of the applied deformations. In (b), (c) and (d), vertices are colored depending on the influence of elbow joint which is mostly deformed. In (a), vertices are colored based on their z coordinate.

language was preferred for a more user and presentation-friendly experience; for a more robust and efficient implementation C++ would be advised.

An instance of a class called v_w is used to store for each vertex a list of up to 4 bones that influence it along with the corresponding influence factors. The node tree is then traversed and all information regarding rotation, translation and dilation are translated to multivectors [8,11] and also stored in the instance for convenience. In order to evaluate the final position of the vertices, all that is left is to evaluate the sum in Eq. (3) for all vertices and down project it to $\mathbb{R}^3$, for each vertex. There are two possible ways of achieving this task. The first way is to evaluate the sum and then down project the final result to obtain each vertex in Euclidean form. The second way is to down project each term and then add them to get the final result. Although not obvious, the second method yields faster results since the addition of 4 multivectors (32-dimensional arrays) and one down-projection is slower than down-projecting (up to) 4 multivectors and adding 4 euclidean vectors of dimension 3.

A final implementation detail regards the weight evaluation for newly added vertices in the cutting and tearing modules. In the former module, such vertices necessarily lie on an edge of the original mesh, whose endpoints both lie on different sides of the cutting plane. Another method is the one used in the tearing module where the intersection point can also lie inside a face. Assuming the point X lie somewhere on the face ABC, we can explicitly write $OX = pOA + qOB + rOC$ for some $a, b, c \in [0, 1]$ such that $p + q + r = 1$. The tuple (p, q, r) is called the *barycentric coordinate* of X with respect to the triangle

ABC. Each of the vertices A, B, C are (usually) influenced by up to 4 bones, so let us consider that they are all influenced by a set of $N (\leq 12)$ vertices, where the bones beside the original 4 have weight 0. Let w_A, w_B, w_C, w_X denote the vectors containing the N weights that correspond to vertices A, B, C and X respectively, for the same ordering of the N involved bones. To determine w_X, we first evaluate $w = p w_A + q w_B + r w_C$ and consider two cases. If w contains up to 4 non-zero weights, then $w_X = w$. Otherwise, since each vertex can be influenced by up to 4 bones, we keep the 4 greater values of w, set the others to zero, and normalize the vector so that the sum of the 4 values add to 1; the final result is returned as w_X. We denote this weight as *weight of X via barycentric coordinates*. Variations of this technique can be applied in both modules to prioritize or neglect influences on vertices lying on a specific side of the cutting plane. Different variations of the weight function allows for less artifacts, depending on the model and the deformation subsequent to the cutting/tearing.

Performance: Running the Tearing algorithm in the arm model (5037 faces, 3069 vertices) it took 2437 ms for the final output, for 34 intersection points. Most of this time (2411 ms) were needed just to determine which two faces were intersected by the scalpel. Tearing the cylinders model (758 faces, 634 vertices) took 362 ms for 17 intersection points. Again, most time (331 ms) was spend for the scalpel intersection. For the Cutting Algorithm, it took for the cylinders model a total of 898 ms: 42 ms for vertex separation, 757 ms for re-triangulation of the 92 intersection points, 87 ms to split faces in two meshes and 12 ms to update the weights. To cut the arm model, it took 22805 ms, where most of them (22547 ms) were spent to re-triangulate the 90 intersection points. These running times can be greatly improved as our current unoptimized CPU-based Python implementation has to thoroughly search all faces for cuts/tears. A GPU implementation optimized for multivector operations would allows to the comparison of our proposed method with the current state-of-the-art methods.

Video: A video with our results can be found at https://bit.ly/3fsYkdZ.

4   Conclusions and Future Work

This work describes a way to perform model animation and deformation as well as cutting and tearing under a single geometric framework called Conformal Geometric Algebra. Our results were obtained using python but since our goal is to have a full implementation in real-time virtual reality simulation we will inevitably have to embed in C++ and ultimately Unity/Unreal Engine code. We intend to combine the tearing module in conjunction with a physics engine to obtain a realistic opening effect. A drilling module is in progress that will allow the user to make holes on the skinned model; such a task is useful especially for VR simulations of dental surgeries. Finally, it is our intention to minimize running times to real-time implementation levels via optimization and the use of recently developed acceleration techniques [10].

References

1. Alexa, M.: Linear combination of transformations. ACM Trans. Graph. **21**(3), 380–387 (2002)
2. Bender, J., Müller, M., Otaduy, M.A., Teschner, M., Macklin, M.: A survey on position-based simulation methods in computer graphics. Comput. Graph. Forum **33**(6), 228–251 (2014)
3. Berndt, I.U., Torchelsen, R.P., Maciel, A.: Efficient surgical cutting with position-based dynamics. IEEE Comput. Graphics Appl. **37**(3), 24–31 (2017)
4. Bielser, D., Glardon, P., Teschner, M., Gross, M.: A state machine for real-time cutting of tetrahedral meshes. In: 11th Pacific Conference on Computer Graphics and Applications, pp. 377–386. IEEE Computer Society (2004)
5. Bielser, D., Maiwald, V.A., Gross, M.H.: Interactive cuts through 3-dimensional soft tissue. Comput. Graph. Forum **18**(3), 31–38 (1999)
6. Bruyns, C.D., Senger, S., Menon, A., Montgomery, K., Wildermuth, S., Boyle, R.: A survey of interactive mesh-cutting techniques and a new method for implementing generalized interactive mesh cutting using virtualtools ‡. J. Vis. Comput. Animation **13**(1), 21–42 (2002)
7. Bruyns, C.D., Senger, S.: Interactive cutting of 3D surface meshes. Comput. Graph. **25**(4), 635–642 (2001)
8. Dorst, L., Fontijne, D., Mann, S.: Geometric algebra for computer science - an object-oriented approach to geometry. The Morgan Kaufmann series in computer graphics (2007)
9. Hadfield, H., Lasenby, J.: Direct Linear Interpolation of Geometric Objects in Conformal Geometric Algebra. Advances in Applied Clifford Algebras (2019)
10. Hadfield, H., Hildenbrand, D., Arsenovic, A.: Gajit: symbolic optimisation and JIT compilation of geometric algebra in python with GAALOP and numba. In: Gavrilova, M., Chang, J., Thalmann, N.M., Hitzer, E., Ishikawa, H. (eds.) CGI 2019. LNCS, vol. 11542, pp. 499–510. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-22514-8_50
11. Hildenbrand, D.: Foundations of Geometric Algebra Computing. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-31794-1
12. Kavan, L., Collins, S., Žára, J., O'Sullivan, C.: Geometric skinning with approximate dual quaternion blending. dl.acm.org 27(4) (2008)
13. Kenwright, B.: A beginners guide to dual-quaternions: What they are, how they work, and how to use them for 3D character hierarchies. In: WSCG 2012 - Conference Proceedings, pp. 1–10. Newcastle University, United Kingdom, December 2012
14. Kim, Y.B., Han, J.H.: Bulging-free dual quaternion skinning. In: Computer Animation and Virtual Worlds, pp. 321–329. Korea University, Seoul, South Korea, John Wiley & Sons, Ltd., January 2014
15. Mor, A.B., Kanade, T.: Modifying soft tissue models: progressive cutting with minimal new element creation. In: Delp, S.L., DiGoia, A.M., Jaramaz, B. (eds.) MICCAI 2000. LNCS, vol. 1935, pp. 598–607. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-540-40899-4_61
16. Müller, M., Chentanez, N., Macklin, M.: Simulating visual geometry. In: Proceedings - Motion in Games 2016: 9th International Conference on Motion in Games, MIG 2016, pp. 31–38 (2016)
17. Papaefthymiou, M., Hildenbrand, D., Papagiannakis, G.: An inclusive Conformal Geometric Algebra GPU animation interpolation and deformation algorithm. Vis. Comput. **32**(6–8), 751–759 (2016)

18. Papagiannakis, G.: Geometric algebra rotors for skinned character animation blending. In: SIGGRAPH Asia 2013 Technical Briefs, SA 2013, December 2013
19. Wareham, R., Cameron, J., Lasenby, J.: Applications of conformal geometric algebra in computer vision and graphics. IWMM/GIAE **3519**(1), 329–349 (2004)
20. Wu, J., Westermann, R., Dick, C.: A survey of physically based simulation of cuts in deformable bodies. Comput. Graph. Forum **34**(6), 161–187 (2015)