CGI2016 manuscript No.
(will be inserted by the editor)

An inclusive Conformal Geometric Algebra GPU animation interpolation and deformation algorithm

Margarita Papaefthymiou, Dietmar Hildenbrand, George Papagiannakis

Abstract In the last years, Geometric Algebra with its Euclidean, Homogeneous and Conformal models attracts the research interest in many areas of Computer Science and Engineering and particularly in Computer Graphics as it is shown that they can produce more efficient and smooth results than other algebras. In this paper, we present an all-inclusive algorithm for real-time animation interpolation and GPU-based geometric skinning of animated, deformable virtual characters using the Conformal model of Geometric Algebra (CGA). We compare our method with standard quaternions, linear algebra matrices and dual-quaternions blending and skinning algorithms and we illustrate how our CGA-GPU inclusive skinning algorithm can provide as smooth and more efficient results as state-of-the-art previous methods. Furthermore, the elements of CGA that handle transformations (CGA motors) can support translation, rotation and dilation(uniform scaling) of joints under a single, GPU-supported mathematical framework and avoid conversion between different mathematical representations in contrast to quaternions and dual-quaternions that support only rotation and rotation-translation, respectively. Hence, our main novelty is the replacement of different types of algebras, and their in-between conversions between CPU and GPU, such as linear algebra matrices, quaternions, dual-quaternions and Euler angles for animation interpolation and skinning with a single mathematical representation, the CGA motors which can optimally handle the composition of translation, rotation and scaling joint transformations and interpolations. Employing latest CGA code generators, we provide a sample implementation of our algorithm running natively in a vertex shader program on modern GPUs for typical deformable virtual character simulations.

Keywords Geometric Algebra, Conformal model, Virtual Reality, Animation Blending, Animation, Skinning, GPU-based skinning, Virtual Character Simulation

Papaefthymiou M.
Foundation for Research and Technology, Hellas
University of Crete, Greece

Hildenbrand D.
Hochschule RheinMain, Germany

Papagiannakis G.
Foundation for Research and Technology, Hellas
University of Crete, Greece

1 Introduction

In this work, we aim to enhance the Conformal model of Geometric Algebra (CGA) [1,2] as the mathematical background for character animation control [3] and particularly for animation blending and GPU-based geometric skinning (character deformation). Geometric Algebra is a mathematical framework that provides a single, convenient all-inclusive algebra for representing orientations and rotations of objects in three or higher dimensions, a compact and geometrically intuitive formulation of transformation algorithms, and an easy and immediate computation of rotors (subsuming quaternions, dual-quaternions, and complex numbers); CGA extends the usefulness of the 3D Geometric Algebra by expanding the class of rotors to include translations and dilations (uniform scaling). Rotors are simpler to manipulate than Euler angles, more numerically stable and more efficient than rotation matrices for the composition of transformations, avoiding the problem of Gimbal lock. Hence, they provide a single mathe-

matical framework ideal for all virtual character animation and deformation typical transformations, without the need to convert from one representation to another. So far though it has not been extensively studied how this framework could be employed directly in modern GPUs, resulting in bypassing it almost entirely so that standard, state-of-the-art game engines would employ at most quaternions or very few dual-quaternions at most as intermediate expressions.

We compare our method with state-of-the-art quaternions and dual-quaternions [4] blending and skinning algorithms and we show that our methodology can provide as smooth and efficient results as quaternions and dual-quaternions. However, dual-quaternions have a limitation: they cannot handle scaling transformations. In contrast, our method can support scaling and scaling interpolation compositions in addition to rotation and translation, under a unique representation by avoiding conversion between different mathematical representations and between CPU and GPU. Our main novelty is the replacement of such different types of algebras like dual-quaternions and matrices for animation blending and skinning with only one mathematical type, the Conformal Geometric Algebra motors which can handle translation, rotation and scale and runs natively in GLSL on modern GPUs in a vertex shader program.

The results of this work allow us to (a) achieve high performance and unify previously separated linear and (dual) quaternion algebra transformation compositions, (b) fully replace quaternions for rotation interpolation with fast CGA rotors and (c) blend rotations, translations and dilations between character animations using CGA, under a single geometric algebraic framework, (d) result in more efficient character simulations for modern gamification and drop-in replacement of existing animation interpolation and skinning for modern game engines.

The paper is organized as follows: The Sect.1 provides previous and related work, the Sect.2 gives a review of Geometric Algebra and the Conformal model of Geometric Algebra mathematical frameworks which are used to implement our algorithms. Sect.3 provides the necessary implementation details, Sects.4 and 5 describe the CGA-GPU animation blending and skinning algorithms, Sect.6 presents our results and a detailed comparison with existing methods. Sect.8 provides our conclusions and future work.

2 Previous work

In recent years, Geometric Algebra and Conformal Geometric Algebra mathematical tools are used in many areas of Computer Science and Computer Engineering, such as Computer Vision, Computer Graphics and Robotics.

Dorst et al. [1] present applications of GA and CGA in the field of Computer Graphics and provide useful examples written in C++ using the Gaigen library and code generator. Some of these applications are interpolating rotations, recursive ray-tracing for illumination, constructing Binary Space Partition (BSP) trees, handling rotations with rotors and handling intersections for collision detection and shadows.

Papagiannakis et al. [3,5] proposed two alternative methodologies for implementing real-time animation interpolation for skinned characters using GA rotors and showed that they achieve smaller computation time, lower memory usage and more visual quality results compared to the other methods. Papaefthymiou et al. [6] proposed a method for handling rotation of the AR objects using Geometric Algebra rotors. Wareham et al. [7] proposed a method for pose and position interpolation using CGA which can also be extended to higher-dimension spaces. Moreover, Wareham et al. [8] proposed a method for interpolating smoothly between two or more displacements that include rotation and translation using CGA. Also, Kavan et al. [4] presented an interpolation method of rotation and translation for skinning using dual-quaternions with fast performance. Thalmann et al. [9] introduced matrix operation approach for skin deformation which overcomes the problem of vertex collapsing. Also, Thalmann et al. [10] proposed Joint-dependent Local Deformation (JLD) operators for moving hands and grasping object which also provides hand deformation while moving.

3 Review of Geometric Algebra

GA [1,2,11,12] is a mathematical framework that provides a convenient mathematical notation for representing orientations and rotations of objects in three dimensions, a compact and geometrically intuitive formulation of algorithms, and an easy and immediate computation of rotors.

3.1 3D Euclidean Geometric Algebra

The basis vectors for the 3-dimensional Euclidean Geometric Algebra space are the orthonormal basis e_1, e_2 and e_3 which are the basic elements for generating the GA. The products of GA are the outer product, the inner product and the geometric product. The outer product, often called wedge product, is denoted by $\wedge$ and is computed with the following equation:

$$\begin{aligned} a \wedge b &= (a_1e_1 + a_2e_2 + a_3e_3) \wedge (b_1e_1 + b_2e_2 + b_3e_3) \\ &= (a_1b_2 - a_2b_1)e_1 \wedge e_2 + (a_2b_3 - a_3b_2) \\ &\quad e_2 \wedge e_3 + (a_3b_1 - a_1b_3)e_3 \wedge e_1 \end{aligned} \tag{1}$$

where a and b are vectors. It can be constructed a higher level dimmensionality oriented subspace by defining the outer product between vectors. Such a subspace is called blade and a k-blade denotes a k-dimensional subspace. For example, a vector is 1-blade, the outer product of 2 vectors is 2-blade, called bivector, the outer product of 3 vectors is 3-blade, called trivector etc. A bivector represents a plane and a trivector represents a 3D volume. The bivectors of the 3D euclidean GA are $e_1 \wedge e_2$, $e_2 \wedge e_3$, $e_3 \wedge e_1$ and the trivector is $e_1 \wedge e_2 \wedge e_3$. The highest blade element is called pseudoscalar and is denoted by I. For example, the pseudoscalar in 3D euclidean space is $I_3 = e_1e_2e_3$.

The inner product, often called dot product is denoted by $\cdot$ and is used to compute distance and angles. The inner product is:

$$a \cdot b = |a||b|cos\phi \tag{2}$$

where ϕ is the angle formed by the vectors a and b.

The geometric product is a mixed grade product: it consists of a scalar which is 0-blade and a bivector which is 2-blade, and is an example of a multivector. The geometric product is:

$$ab = a \cdot b + a \wedge b = |a||b|(cos\phi + Isin\phi) = |a||b|e^{I\phi} \tag{3}$$

The duality of a GA element is denoted by $*$ and gives a blade that represents the orthogonal complement of that subspase. For example, the duality of a bivector equals to the vector that is perpendicular to this bivector and vice versa. The duality is defined with the Equation below:

$$A^* = A/I = -AI \tag{4}$$

As an example below the duality of the basis vector in the 3D space is computed as follows:

$$\begin{aligned} e_2^* &= -e_2I_3 = -e_2(e_1e_2e_3) \\ &= e_2e_1e_2e_3 = -e_2e_2e_1e_3 = -e_1e_3 = -e_1 \wedge e_3 \end{aligned} \tag{5}$$

The basic element used to handle rotations of any multivector in GA is rotor and is usually denoted as R and is computed using the exponential Formula below:

$$R = e^{-I_3u\frac{\phi}{2}} \tag{6}$$

where ϕ is the angle of rotation and u is the axis of rotation. The interpolated rotation between two rotors R_1 and R_2 in N steps is given by:

$$R_N = e^{log(R_2R_1^{-1})*N} \tag{7}$$

To rotate a multivector A we sandwich it between the rotor R and its inverse rotor R^{-1} as shown below:

$$RAR^{-1} \tag{8}$$

3.2 Conformal Geometric Algebra

CGA [13] is a 5D space algebra which is able to handle 3D transformations (Conformal transformations) like translations, rotations and dilations by expanding the 3D Euclidean Geometric Algebra with two additional basis vectors. These additional basis vectors are e_- and e_+, which have opposite signatures are defined such that:

$${e_+}^2 = 1, \quad {e_-}^2 = -1, \quad e_+ \cdot e_- = 0$$

Using the basis vector e_- and e_+, are defined two additional basis vectors e_0 the 3D point at the origin and e_∞ the infinity point which are null:

$$e_0^2 = 0, \quad e_\infty^2 = 0$$

and are constructed as follows:

$$e_\infty = e_- + e_+, \quad e_0 = \tfrac{1}{2}(e_- - e_+)$$

3.2.1 Representing Entities in CGA

Conformal Geometric Algebra is able to represent basic 3D primitives [14] using inner and outer products which are a)Point, b)Sphere, c)Plane, d)Circle, e)Line and f) Point pair.

Point: 3D points in the conformal space are extended in 5D space using the following Equation:

$$P = x + \frac{1}{2}x^2e_\infty + e_0 \tag{9}$$

where x is the 3D point in the 3D Euclidean GA model.

Sphere: A sphere can be represented using its radius r and its center point P:

$$S = P - \frac{1}{2}r^2 e_\infty \tag{10}$$

Alternatively, is constructed with the outer product of four points that lie on the sphere:

$$S^* = P_1 \wedge P_2 \wedge P_3 \wedge P_4 \tag{11}$$

Plane: A plane is constructed as follows:

$$\pi = n + de_\infty \tag{12}$$

where where n is the 3D normal vector on the plane is the normal on plane π and d is the distance from the origin. A plane can also constructed with the outer product of three points that lie on it and infinity point.

$$\pi^* = P_1 \wedge P_2 \wedge P_3 \wedge e_\infty \tag{13}$$

Circle: A circle is defined with the intersection of two spheres $S1$ and $S2$:

$$Z = S_1 \wedge S_2 \tag{14}$$

or using three points that lie on it:

$$Z^* = P_1 \wedge P_2 \wedge P_3 \tag{15}$$

Line: A line is constructed with the intersection of two planes

$$L = \pi_1 \wedge \pi_2 \tag{16}$$

or with the use of two points that lie on it and the point at infinity

$$L^* = P_1 \wedge P_2 \wedge e_\infty \tag{17}$$

Point pair: A point pair is defined with the help of two points:

$$P^* = P_1 \wedge P_2 \tag{18}$$

An alternative way, is with the intersection of three spheres:

$$P = S_1 \wedge S_2 \wedge S_3 \tag{19}$$

3.2.2 Transformations in CGA

Translators: In the Conformal space we can translate a vector x with a Translator rotor:

$$T = e^{-\frac{1}{2}te_\infty} = 1 - \frac{1}{2}te_\infty \tag{20}$$

where t is the vector that represents translation:

$$t = t_1e_1 + t_2e_2 + t_3e_3 \tag{21}$$

Rotors: Rotor operator in CGA space is constructed using the exponential Equation:

$$R = e^{-b\frac{\phi}{2}} = e^{-I_3u\frac{\phi}{2}} = cos(\frac{\phi}{2}) - uI_3sin(\frac{\phi}{2}) \tag{22}$$

where ϕ is the angle of rotation, b is the plane of rotation and u is the axis of rotation.

Dilators: Dilation operator gives the scaling of factor d about the origin e_0 using the formula below:

$$D = 1 + \frac{1-d}{1+d}e_\infty \wedge e_0 \tag{23}$$

Motors: In CGA a transformation that includes rotation and translation, called displacement versor or motor is given by:

$$M = RT \tag{24}$$

where R is the rotor and T is the translator. The following formula is used to linearly interpolate between two motors M_1 and M_2 in N steps:

$$M_N = M_1 * (1 - N) + M_2 * N \tag{25}$$

The following Equation is used to apply the motion to a rigid body A, where M^{-1} is the inverse of the motor M:

$$MAM^{-1} \tag{26}$$

3.3 Representing Quaternions and Dual-Quaternions with Geometric Algebra

Quaternions of the form:

$$Q = 1 + (xi + yj + zk)$$

are represented in Conformal Geometric Algebra based on four blades: the scalar and three two-blades, that is all even grade elements in $Cl(R^3)$. Dual-Quaternions of the form:

$$\begin{aligned} Q &= Q_1 + \epsilon Q_2 \\ &= 1 + (x_1 i + y_1 j + z_1 k) + \epsilon(1 + (x_2 i + y_2 j + z_2 k)) \\ &= 1 + (x_1 i + y_1 j + z_1 k) + (\epsilon + x_2 i\epsilon + y_2 j\epsilon + z_2 k\epsilon) \end{aligned}$$

where Q_1 and Q_2 are Quaternions, are represented in CGA based on eight blades: the scalar, six two-blades and one null four-blade ϵ.

Table 1 shows the correspondence of Quaternions and Dual-Quaternions with CGA blades.

Quaternion	Dual-Quaternion	CGA
1	1	e
i	i	$e_1 \wedge e_2$
j	j	$e_3 \wedge e_1$
k	k	$e_2 \wedge e_3$
$-$	$k\epsilon$	$e_\infty \wedge e_1$
$-$	$j\epsilon$	$e_\infty \wedge e_2$
$-$	$i\epsilon$	$e_\infty \wedge e_3$
$-$	ϵ	$e_1 \wedge e_2 \wedge e_3 \wedge e_\infty$

Table 1: Representing Quaternions and Dual-Quaternions in CGA.

4 Implementation Details

To implement our algorithms we have used the following open-source APIs and libraries: a) the ASSIMP import library for loading our skinned characters b) GLM mathematical library for quaternions and dual-quaternions interpolation c) OpenGL API version 3.2 for real-time rendering and GLSL version 1.5.

For Geometric Algebra and Conformal Geometric Algebra operations we have used Gaalop Precompiler [15] to generate C++ code. Gaalop Precompiler optimizes Geometric Algebra code which is written in CluCALC scripting language in order to achieve higher performance. To generate our algorithms and visualize their results we have used the CLUCalc v4.3 visualization tool [16]. To generate C++ code we have used Gaalop standalone application with Gaalop Precompiler. Gaalop precompiler supports three types of optimization: GAPP (Geometric Algebra Parallelism Program), Maple and Table-Based Approach. For our experiments provided in Section 7 we have used Table-Based Approach.

5 Animation Interpolation Algorithm description

Our main novelty, is the employment of Conformal GA motors as fast, drop-in replacements for quaternion algebra and dual quaternion algebra, during animation blending for skinned characters. On the following approach we represent rotation combined with translation and dilation with CGA motors and we use linear interpolation to interpolate the motors between the two keyframes of the character animation.

5.1 CGA motors approach

In this approach, the rotation, translation and scaling of the two keyframes of the animation are represented as Conformal Geometric Algebra Motors representation. ASSIMP library provides the rotation of the animation in quaternion representation. In order to avoid converting quaternion to CGA rotor on each frame, we add an additional field to the structure of the bone (CGA rotor field) of the ASSIMP library and we precompute quaternion as CGA rotor representation. The next steps show how we convert rotation and translation to motor representation using CluCALC scripting language:

1. Convert translation vector to 3D euclidean geometric algebra vector with the help of basis vectors e_1, e_2 and e_3 where (transX,transY,transZ) is the euclidean translation vector :
   translationGA=transX*e1+transY*e2+transZ*e3;
2. Convert 3D GA vector (translationGA) to CGA translator rotor using the Equation 20 where einf is the infinity point:
   translationCGA=1-0.5*translationGA*einf;
3. Compute the angle and the axis of the quaternion using glm mathematical library functions:
   float angle=angle(quaternion);
   vec3 axis=axis(quaternion);
4. Similarly with step 1, we convert the axis (axis.x,axis.y,axis.z) of the quaternion to a 3D GA point:
   vector=VecN3(axis1X,axis1Y,axis1Z);
5. Construct a line that represents the axis of the quaternion using the Equation 16:
   axis = vector∧VecN3(0,0,0)∧einf;
6. Compute the duality of the line e.g the plane of rotation (bivector) (Equation 4):
   plane=*axis;

7. Construct rotor representation (Equation 22):
   R=exp(-0.5*angle*plane);
8. Motor is computed as the geometric product of the translator and rotor (Equation 26):
   motor=translationCGA*R;

After expressing the source and destination rotation-translation to CGA motors we interpolate them base on a factor number that defines the animation interpolation step. We compute the motor from source (M_{src}) to destination (M_{dst}) motor in N steps using the linear Equation 25, which is written in CluCALC script as follows:
interpolated = Msrc*(1-alpha)+Mdst*alpha;

To construct the dilator of a key frame we use the Equation 23 which is written in CluCALC script as follows:
Dilator = 1+ (1-d1)/(1+d1)*einf∧e0;

where d1 is the scaling of the keyframe. To interpolate between two dilators we use the following CluCALC script formula:
finalD =Dilator + (Dilator2-Dilator)*alpha;

where Dilator is the source dilator, Dilator2 is the destination dilator and alpha the factor of interpolation.

Final transformation is computed by the geometric product of motor and dilator as follows:

$$final = M_{final} * finalD \tag{27}$$

On Section A.1 we provide the overall CLUCalc code for converting keyframes transformation to CGA motors and interpolate them.

6 GPU-based skinning Algorithm description

In our method, we handle geometric skinning using CGA motors in the vertex shader. To apply transformation on the vertices we sandwiching the position of the vertex between the motor and it's inverse motor (Equation 26). We generate code that transforms a point using motor that comprises of rotation, translation and dilation using CluCALC scripting language as follows:

motor=translation*rotation*dilation;
newPoint=motor*VecN3(x,y,z)* ∼motor;

where translation, rotation and dilation are computed as described in Section 5.1

Gaalop precompiler generates motor as a 12 element vector which we convert it to a matrix representation of type mat3x4 (build-in type of GLSL) in order to be able to use it in the vertex shader. In the vertex shader we are declaring uniform array which comprises of the bones' transformations of type mat3x4.

Each vertex of the skinned character may be influenced by four bones each one with different weight. We transform the vertex with each bone separately using the code generated from gaalop precompiler:

```
vec4 pos1=transformP(BonesVersors[BoneIDs[0]],
                     pos.x,pos.y,pos.z);
vec4 pos2=transformP(BonesVersors[BoneIDs[1]],
                     pos.x,pos.y,pos.z);
vec4 pos3=transformP(BonesVersors[BoneIDs[2]],
                     pos.x,pos.y,pos.z);
vec4 pos4=transformP(BonesVersors[BoneIDs[3]],
                     pos.x,pos.y,pos.z);
```

and we compute their weighted average base on the weight of each bone:

```
vec4 finalpos=pos1*Weights[0] + pos2*Weights[1]
             +pos3*Weights[2] + pos4*Weights[3];
```

Section A.2 provides the vertex shader used for skinning with CGA motors.

7 Results

In this section, we compare our animation blending and GPU-based skinning approach with quaternions and dual-quaternions. We obtained the following results using the Platform described in Table 2. The characters used to obtain the results are of dae format, loaded with Assimp library and consist of 42-54 joints and 3851-14985 triangles.

Figures 1, 2, 3 show the time in msecs for each animation blending method and the average frame rate (fps) for animation blending and skinning for three different characters. The first method is quaternions that support rotation combined with translation matrices, the second one is dual quaternions which combine translation and rotation and the third one is our method "CGA-GPU inclusive algorithm" which combines rotation, translation and scaling. On Figures 1, 2, 3 we also, present CGA-GPU inclusive algorithm with only rotation and translation in order to be comparable with quaternions and dual quaternions. As concerning our method with dilation, Character 2 and 3, have identity dilation and only the dilation of Character 1 is non-identity and that is why is slower than the method with out dilation. Our results show that our algorithm is equally efficient with quaternions and dual-quaternions in terms of performance and frame rate but is superior in terms of single mathematical algebraic representation in CPU and GPU. Using our algorithm we

haven't noticed any significant errors during the necessary conversions between different mathematical representations. Table 3, summarizes the contribution of our CGA-GPU inclusive algorithm comparing to Euler-angles / transformation matrices, Quaternions and Dual-Quaternions.

Platform	OS X 10.11.3
Processor	2.5 GHz Intel Core i7
Graphics Card	NVIDIA Geforce GT 750M 2048 MB
Compiler	LLVM 7.0

Table 2: Platform characteristics used to run our experiments.

8 Conclusions and Future work

In this work, we have presented an animation blending method and a GPU-based skinning algorithm using Conformal Geometric Algebra. We handle transformations using CGA motors that combine to a unique representation translation, rotation and scaling.

Method	Time (ms)	Frame Rate (fps)
Quaternions (rotations only)	0.0004	59.6
Dual Quaternions (rotations and translation)	0.0003	59.6
CGA-GPU inclusive algorithm (rotations, translations)	0.0003	59.6
CGA-GPU inclusive algorithm (rotations, translations and dilations (scaling))	0.0005	59.6

Fig. 1: Comparison of animation blending of time in msecs of Quaternions, Dual-Quaternions, CGA-GPU inclusive algorithm and the average frame rate (fps) for animation blending and skinning on 1st Character (PolygonCount: 2548, Animation length:1.08333 ticks, Number of bones: 52)

Method	Time (ms)	Frame Rate (fps)
Quaternions (rotations only)	0.0017	59.6
Dual Quaternions (rotations and translation)	0.0016	59.6
CGA-GPU inclusive algorithm (rotations, translations)	0.0017	59.6
CGA-GPU inclusive algorithm (rotations, translations and dilations (scaling))	0.0022	59.6

Fig. 2: Comparison of animation blending of time in msecs of Quaternions, Dual-Quaternions, CGA-GPU inclusive algorithm and the average frame rate (fps) for animation blending and skinning on 2nd Character (PolygonCount: 135976, Animation length:12.3667 ticks, Number of bones: 54)

Method	Time (ms)	Frame Rate (fps)
Quaternions (rotations only)	0.0012	59.6
Dual Quaternions (rotations and translation)	0.0009	59.6
CGA-GPU inclusive algorithm (rotations, translations)	0.0012	59.6
CGA-GPU inclusive algorithm (rotations, translations and dilations (scaling))	0.0015	59.6

Fig. 3: Comparison of animation blending of time in msecs of Quaternions, Dual-Quaternions, CGA-GPU inclusive algorithm and the average frame rate (fps) for animation blending and skinning on 3rd Character (PolygonCount: 14985, Animation length:8.33333 ticks, Number of bones:43)

Methodology	Rotation	Translation	Dilation	Performance	Single Representation
Euler-angles/ Transformation matrices	✓	matrices only	matrices only	Gimbal lock, cannot interpolate-need to be in other representation	Transformation matrices only
Quaternions	✓	-	-	Interpolated only at the origin, need conversion to transformation matrix to transform point	no translation no dilation
Dual-Quaternions	✓	✓	-	Interpolated only at the origin, need conversion to transformation matrix to transform point	no dilation
Our method:CGA-GPU inclusive algorithm	✓	✓	✓	Efficient performance, interpolate around any axis, can transform any entity	✓

Table 3: General comparison of Euler-angles/Transformation matrices, Quaternions, Dual-Quaternions and our method:CGA-GPU inclusive algorithm.

Our results show that our method achieves high performance and smooth results as well as quaternions and dual-quaternions. However, our method allows handling blending and skinning without needed to use any other algebraic frameworks in contrast to quaternions and dual-quaternions that can handle only rotations and rotations-translations respectively.

In the future, we aim to experiment our animation blending and GPU-based skinning and to other platforms (e.g. Windows) since MacOSX El Capitan 10.11 does not allow to disable vsync. We also aim to achieve interpolation using logarithms of motors for rotation, translation and scaling in the vertex shader. Lastly, we intend to extend our CGA framework by applying GA for global illumination and specifically, for rotating spherical harmonics for Precomputed Radiance Transfer for real-time rendering.

Acknowledgements

The research leading to these results has received funding from the European Union People Programme (FP7-PEOPLE-2013-ITN) under grant agreement n^O 608013.

References

1. L. Dorst, D. Fontijne, and Mann S. *Geometric Algebra for Computer Science*. Morgan Kaufmann, 2007.
2. D. Hestens and G. Sobczyk. *Clifford Algebra to Geometric Calculus: A Unified Language for Mathematics and Physics (Fundamental Theories of Physics)*. Springer, 1984.
3. G. Papagiannakis, E. Greasidou, P. Trahanias, and M. Tsioumas. Mixed-reality geometric algebra animation methods for gamified intangible heritage. *International Journal of Heritage in the Digital Era*, 3:683–699, Apr. 2014.
4. L. Kavan, S. Collins, J. Žára, and C. O'Sullivan. Geometric skinning with approximate dual quaternion blending. *ACM Trans. Graph.*, 27(4):105:1–105:23, November 2008.
5. G. Papagiannakis. Geometric algebra rotors for skinned character animation blending. In *SIGGRAPH Asia 2013 Technical Briefs*, SA '13, pages 11:1–11:6, New York, NY, USA, 2013. ACM.
6. Margarita Papaefthymiou, Andrew Feng, Ari Shapiro, and George Papagiannakis. A fast and robust pipeline for populating mobile ar scenes with gamified virtual characters. In *SIGGRAPH Asia 2015 Mobile Graphics and Interactive Applications*, SA '15, pages 22:1–22:8, New York, NY, USA, 2015. ACM.
7. R. Wareham, J. Cameron, and J. Lasenby. Applications of conformal geometric algebra in computer vision and graphics. In Hongbo Li, Peter J. Olver, and Gerald Sommer, editors, *IWMM/GIAE*, volume 3519 of *Lecture Notes in Computer Science*, pages 329–349. Springer, 2004.
8. R. Wareham and J. Lasenby. Mesh vertex pose and position interpolation using geometric algebra. In *Articulated Motion and Deformable Objects, 5th International Conference, AMDO 2008, Port d'Andratx, Mallorca, Spain, July 9-11, 2008, Proceedings*, pages 122–131, 2008.
9. N. Magnenat-Thalmann, F. Cordier, Hyewon Seo, and G. Papagianakis. Modeling of bodies and clothes for virtual environments. In *Cyberworlds, 2004 International Conference on*, pages 201–208, Nov 2004.
10. N. Magnenat-Thalmann, R. Laperrière, and D. Thalmann. Joint-dependent local deformations for hand animation and object grasping. In *Proceedings on Graphics Interface '88*, pages 26–33, Toronto, Ont., Canada, Canada, 1988. Canadian Information Processing Society.
11. K. Kanatani. *Understanding Geometric Algebra: Hamilton, Grassmann, and Clifford for Computer Vision and Graphics*. A K Peters/CRC Press, 2015.
12. Hitzer E. Introduction to clifford 's geometric algebra. *SICE Journal of Control, Measurement, and System Integration*, 4:001–011, 2011.
13. G. Sommer. *Geometric computing with Clifford algebras: theoretical foundations and applications in computer vision and robotics*. Springer London, 2001.
14. Eckhard Hitzer, Kanta Tachibana, Sven Buchholz, and Isseki Yu. Carrier method for the general evaluation and control of pose, molecular conformation, tracking, and the like. *Advances in Applied Clifford Algebras*, 19(2):339–364, 2009.
15. D. Hildenbrand. *Foundations of Geometric Algebra Computing*, volume 8. Springer, 2013.
16. C. Perwass. *Geometric Algebra with Applications in Engineerings*. Springer, 2009.

Appendix A CGA-GPU inclusive algorithm

A.1 CLUCalc implementation for Animation Blending with CGA motors approach.

```
//exponential
Exp_approx = { 1 + _P(1) + _P(1)*_P(1)/2 + _P(1)*_P(1)*_P(1)/6 + _P(1)*_P(1)*_P(1)*_P(1)/24 };

//source translation
translationGA=transX*e1+transY*e2+transZ*e3;
translationCGA=1-0.5*translationGA*einf;
vector=VecN3(axis1X,axis1Y,axis1Z);
plane = *(vector∧VecN3(0,0,0)∧einf);
//source rotation
R = Exp_approx(-angle1/2*plane);
//source dilation
Dilation = 1+ (1-d1)/(1+d1)*einf∧e0;

//destination translation
translationGA2=trans2X*e1+trans2Y*e2+trans2Z*e3;
translationCGA2=1-0.5*translationGA2*einf;
vector2=VecN3(axis2X,axis2Y,axis2Z);
plane2 = *(vector2∧VecN3(0,0,0)∧einf);
//destination rotation
R2 = Exp_approx(-angle2/2*plane2);
//destination dilation
Dilation2 = 1+ (1-d2)/(1+d2)*einf∧e0;

//source and destination motor
motor1=translationCGA*R;
motor2=translationCGA2*R2;

//linear interpolation of motors
interpolatedTR=motor1*(1-alpha)+motor2*alpha;

//linear interpolation of dilations
interpolatedD =Dilation + (Dilation2-Dilation)*alpha;

//final transformation
finalInterpolation=interpolatedTR*interpolatedD;

//transform a point
vertexPos=finalInterpolation*VecN3(X,Y,Z)* finalInterpolation;
// In homogeneous coordinates x = e1 blade, y = e2 blade, z = e3 blade, w = e0 blade
```

A.2 Vertex Shader

```
#version 330 core

// input: vertex position
in vec3 vPosition;
// input: texture coordinate
in vec2 vTexCoord;

// each vertex influenced from 4 bones
//input: boneIDs which influence vertex
in ivec4 BoneIDs;
//input: weight of influence
in vec4 Weights;

// uniform variables: projection and modelview matrix
// no need to convert in CGA motor representation - transform vertex position directly
uniform mat4 ModelView;
uniform mat4 Projection;

// max bones of the character
const int MAX_BONES = 100;
// transformation of each bone in CGA motor representation
// the CGA motor (12 element matrix generated from gaalop) in mat3x4 representation
uniform mat3x4 BonesVersors[MAX_BONES];

//output: texture coordinates
out vec2 texCoord;

vec4 transformP(mat3x4 bones, float X, float Y, float Z)
{
	// transform point with CGA motors using C++ code generated with Gaalop Precompiler (Section A.1)
}

void main()
{
	// transform the vertex with CGA motors using 4 bones
	vec4 pos1 = transformP(BonesVersors[BoneIDs[0]], vPosition.x, vPosition.y, vPosition.z);
	vec4 pos2 = transformP(BonesVersors[BoneIDs[1]], vPosition.x, vPosition.y, vPosition.z);
	vec4 pos3 = transformP(BonesVersors[BoneIDs[2]], vPosition.x, vPosition.y, vPosition.z);
	vec4 pos4 = transformP(BonesVersors[BoneIDs[3]], vPosition.x, vPosition.y, vPosition.z);

	//weighted average of the vertex positions
	vec4 finalPos = pos1*Weights[0]+pos2*Weights[1]+pos3*Weights[2]+pos4*Weights[3];

	//final position= projection * modelview * vertex position after bones transformation
	gl_Position = Projection * ModelView * finalPos;

	// pass to fragment shader per-vertex tex coords
	texCoord = vTexCoord;
}
```