Geometric algebra rotors for skinned character animation blending

briefs_0080*

QLB: 320 FPS GA: 381 FPS QLB: 301 FPS GA: 341 FPS DQB: 290 FPS GA: 325 FPS

Figure 1: *Comparison between animation blending techniques for skinned characters with variable complexity: a) quaternion linear blending (QLB) and dual-quaternion slerp-based interpolation (DQB) during real-time rigged animation, and b) our faster geometric algebra (GA) rotors in Euclidean 3D space as a first step for further character-simulation related operations and transformations. We employ geometric algebra as a single algebraic framework unifying previous separate linear and (dual) quaternion algebras.*

Abstract

The main goal and contribution of this work is to show that (automatically generated) computer implementations of geometric algebra (GA) can perform at a faster level compared to standard (dual) quaternion geometry implementations for real-time character animation blending. By this we mean that if some piece of geometry (e.g. Quaternions) is implemented through geometric algebra, the result is as efficient in terms of visual quality and even faster (in terms of computation time and memory usage) as the traditional quaternion and dual quaternion algebra implementation. This should be so even without taking into account certain algorithmic enhancements that geometric algebra may allow in selected applications. This work describes two implementation approaches for quaternion interpolation using Euclidean GA rotors for skinned character animation blending. It also lays the foundation so that GA can be employed for further calculations (skinning, rendering) under a unified geometry computation framework.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism – Animation;

Keywords: geometric algebra, animation blending, rotor-based interpolation

1 Introduction and Main Concept

Our work focuses on a simple and robust method to interpolate between orientations for rigged, animated articulated figures using geometric algebra (GA) rotors. Our approach aims to devise a novel, integrated framework as alternative to current fragmented algebraic methods used in orientation interpolation (matrices, quaternions, dual quaternions). With GA rotors we aim to provide a drop-in replacement and faster approach for existing real-time animation pipelines. As a result we can interpolate orientations of k-dimensional subspaces using the same algebra, in a non-computationally expensive manner with faster performance.

2 Previous work

[McCarthy 1990] has already illustrated the connection between quaternions and GA bivectors as well as the different Clifford algebras with degenerate scalar products that can be used to describe dual quaternions. Even simple quaternions are identified as 3-D Euclidean taken out of their propert geometric algebra context.

[Fontijne and Dorst 2003] and [Dorst et al. 2007] have illustrated the use of all three GA models with applications from computer vision, animation as well as a basic recursive ray-tracer. [Wareham and Lasenby 2008] employed the GA conformal model (refer to next section for an explanation of the different GA models) to encode position and pose interpolation for skinning. In this work we compare previous quaternion algebraic models with the euclidean GA model as a fast and robust alternative forward-looking representation (more on this in future work).

The use of dual quaternions for enhanced character skinning was primarily been employed by [Kavan et al. 2008] in computer graphics and later by [Chaudhuri et al. 2008] for character animation blending. In [Magnenat-Thalmann et al. 2004] the matrix operator and linear algebra was employed in order to enhance skinning and vertex collapse during animation blending.

In this paper we aim to verify that as quaternions and dual quaternions can be expressed as GA rotors (although the later is not explored in this work) and that they can be equally efficient and computationally viable as alternatives for character animation blending.

3 The Euclidean Geometric Algebra model

Geometric algebra (GA) is a powerful and practical computational framework for the representation and solution of geometrical problems. Its roots can be traced to the 19th century and mainly the work of W. Clifford that unified previous algebras from Grassmann and Hamilton [McCarthy 1990], [Dorst et al. 2007].

* e-mail: briefs_0080

The basic computational elements in GA are *subspaces* and various *products* involving them. The most basic real, m-dimensional linear space V^m, contains in 3D (V^3) the basis vectors $\{e_1, e_2, e_3\}$ and is called the Euclidean, *Vector* space 3D GA model. There is also the 4D vector *Homogeneous* GA model and the 5D vector *Conformal* GA model for which the reader might refer to [Dorst et al. 2007] for complete coverage.

Although coordinates are necessary for input/output, computations in GA are performed directly on subspaces, in a coordinate-free manner. In this work we focus on the Euclidean, Vector GA model whose vectors are characterized by the same attitude and magnitude algebraic properties as in linear algebra (LA) and can be multiplied to produce a scalar using the *inner* product: $e_1 \cdot e_2$. In GA, a common way to construct a higher dimensional oriented subspace from vectors is to use a product that constructs the span of vectors called the *outer* (or wedge) product and denoted by ^. Such a subspace is called a *blade* and the term *k-blade* is used for a k-dimensional subspace. E.g. a vector is a 1-blade, an area is a 2-blade, a volume a 3-blade etc. where *1,2,3* are referred as grade. The highest grade element is called *pseudoscalar* and often denoted as *I*.

To compute a^b for two vectors *a* and *b* (forming a plane) with scalar coefficients α, β:

$$a \wedge b = (\alpha_1 e_1 + \alpha_2 e_2 + \alpha_3 e_3) \wedge (\beta_1 e_1 + \beta_2 e_2 + \beta_3 e_3) = \\ (\alpha_1\beta_2 - \alpha_2\beta_1) e_1 \wedge e_2 + (\alpha_2\beta_3 - \alpha_3\beta_2) e_2 \wedge e_3 + (\alpha_3\beta_1 - \alpha_1\beta_3) e_3 \wedge e_1 \quad (1)$$

Hence any 2-blade in 3D space is decomposed onto a basis of 3 elements (also called *bivectors*): $\{e_1 \wedge e_2, e_2 \wedge e_3, e_3 \wedge e_1\}$. In a similar manner the outer product of 3 vectors results in a trivector subspace $\{e_1 \wedge e_2 \wedge e_3\}$. This space of scalars, vectors, bivectors and trivectors with + and ^ as operations is called the Grassmann algebra of 3D space.

Clifford introduced a new product, the geometric product that unified inner and outer products and formed the basis of GA, encompassing also the quaternions of Hamilton as basic algebraic elements (rotors) and not a separate, special case. The geometric product between the previous vectors *a, b* in a common plane with unit 2-blade *I* is defined as: $ab = a \cdot b + a \wedge b$ which is an element of mixed grade (a 0-grade scalar accompanied by a 2-blade where the whole is called a multivector). The geometric product is the cornerstone in reflection and rotation calculations, especially in its exponential representation (with ϕ the angle from *a* to *b*):

$$ab = a \cdot b + a \wedge b = |a||b|(\cos\phi + I \sin\phi) = |a||b| e^{I\varphi} \quad (2)$$

The pseudoscalar *I* is also often used to indicate duality in GA, so that for a given element A, its dual is denoted as $A^* = IA$, e.g. the $e^*_1 = e_{123}e_1 = e_{23}$ (where the geometric product of e_1, e_2, e_3: $e_1e_2e_3 = e_{123}$ and $e^2_i = 1$). GA has a special way to represent orthogonal transformations, more powerful than using orthogonal matrices: sandwitching a multivector between $R=ab$ and its inverse, this R is called a rotor, and will be mostly used in the following section.

4 Main algorithm

Our main novelty lies in the employment of Euclidean GA rotors as fast, drop-in replacements for quaternion algebra, during animation orientation interpolation. We provide two different approaches to express quaternions as GA rotors and subsequently utilize the exponential rotor formula of Euclidean 3D GA as alternative to spherical quaternion interpolation as shown in the table below:

Table 1 *Two methods to replace quaternions with GA rotors*

```
%Method 1 to express a quaternion as GA Rotor
(MATLAB code):
srcQ = quaternion(0,0,1,1)
srcQ = srcQ.normalize
srcRotor = srcQ.e(1)+srcQ.e(2)*e1+
srcQ.e(3)*e2+srcQ.e(4)*e3
u=axisSrc(1)*e1+axisSrc(2)*e2+axisSrc(3)*e3
Rsrc = gexp(-I3*u*angleSrc/2)
```

```
%Method 2 to express a quaternion as GA
%Rotor (MATLAB code):
Rsrc = srcQ.e(1) + srcQ.e(2)*(e2^e3) -
srcQ.e(3)*(e1^e3) + srcQ.e(4)*(e1^e2)
```

After expressing the existing input and output quaternions as GA rotors, we were able to fast interpolate between them using the rotor exponential formula that allows to interpolate between two orientations R_A and R_B, *via their geometric product,* using *n* interpolation steps, on an axis of rotation α and the Euclidean 3D GA pseudoscalar I_3 over and angle ϕ:

$$R_A R^n = R_B \Rightarrow R = e^{-I_3 a \frac{\phi}{(2n)}} \quad (3)$$

4.1 Algorithm Pipeline

Step1:

- Input: quaternion orientation of existing animation & skinning frameworks
- Output: express with different ways input and output quaternions as GA rotors and members of the algebra:

$$i = e1I3 = e2 \wedge e3 \\ j = e2I3 = -e1 \wedge e3 \\ k = e3I3 = e1 \wedge e2 \\ ijk = -1 \\ q = 1 + i + j + k \quad (4)$$

- $$R = ba = b \cdot a + b \wedge a = \cos\phi - I \sin\phi \Rightarrow \\ R = e^{-I\phi} \Rightarrow R = e^{-I_3 u \frac{\phi}{2}} \quad (5)$$

Step2:

- Input: source and destination Rotors in GA
- Output: interpolate between them using the closed-form formula in Euclidean 3D space (black interpolated GA vectors in the figures in APPENDIX A) and code in **Table 1** above.

Step3:

- Input: interpolated GA rotors
- Output: translate interpolated rotor to matrix or quaternion for real-time rendering of rigged characters (results shown in Figure 1: left image shows standard (dual) quaternion interpolation, right image our drop-in GA rotor replacement) based on code as shown in the Table 2 below.

Table 2 *GA Rotor interpolation*

```
%GA Rotor interpolation (MATLAB code):
Rtot = Rdest/Rsrc
for i=0 : 0.125 : factor
       Ri = gexp((i/2) *sLog(Rtot))
       Rint = (Ri * srcRotor / Ri)
       draw(Rint, 'k')
end
```

5 Main novelties

In this work we aim to address the research questions: a) can GA rotors be more efficient and faster than quaternions? b) Why do we need GA rotors when we can just use quaternions and slerp? Our work so far has illustrated the following comparative advantages of GA rotors:

a) Quaternions can be used only on other quaternions. GA rotors are universal operators capable of rotating other subspaces: lines, planes, and volumes as first class operators. E.g. To rotate a line with a quaternion, you have to convert the quaternion to a matrix. In GA you can represent the quaternion as rotor and directly rotate the line, plane, volume or other subspace with faster results.

b) The $e^{(\phi\, l^* \alpha/2n)}$ from eq. (3) can convert any axis α (not only at the origin) into a rotational operator. Quaternions can be only interpolated via slerp only at the origin.

c) Blending of motions can be done by blending only the logarithms of rotors from Eq. (3)

6 Implementation

We have employed fully open-source APIs and s/w libraries for all parts of the experiments and case studies: a) the OpenGL 3.2 core profile was used for real-time rendering, the Collada 3D file format for skinned, animated virtual characters, c) the ASSIMP library for asset loading, d) the GABLE GA toolkit for mathematical simulation and the Gaigen & libGASandbox libraries for GA expressions in modern C++, using the LLVM compiler. For h/w we utilised a MacBook Pro with a 2.7GHz Intel i7 and an NVIDIA Geforce GT 650M graphics card. We employed three different articulated figures with variable skeleton joints and polygonal complexity (from 3 joints and 4 triangles up to 42 joints and 30000 triangles) and from a simple 6 vertex plane to a standard md2 and collada characters (freely available online). From the sample results illustrated in Figure 1 is evident that our method is indeed faster than current separate linear algebra (matrix based) and quaternion algebras as we avoid the step of converting back and forth between these two separate algebras. Ofcourse in the end of each frame we still need to convert to a matrix notation so that the GPU linear-blend skinning [Magnenat-Thalmann et al. 2004] vertex shader can convert the rotor to vertex coordinates. As a result of the interpolation steps '*n*' in Eq. (3), we can increase this term to utilise more interpolation steps for smoother animation, without any significant performance penalty (this is reported as GAFactor in the accompanying video).

In the AppendixA we provide MATLAB examples using standard, open-source toolboxes for quaternion and GA operations (GABLE) as well as corresponding C++ code in AppendixB based on open-source C++ libraries for real-time, skinned character animation blending (GLM and libGASandbox).

4 Conclusions and future work

In this work we have scratched the surface of employing GA rotors to modern character simulation frameworks in order to:

- Unify and improve the performance of previouly separated linear and quaternion algebra transformations under a single geometric algebraic framework
- fully replace quaternions for skinning/animation with faster Euclidean GA rotors
- blend between skeleton animations using GA

Our future work involves extending the above framework by expressing dual-quaternions in either the Euclidean 3D or the Conformal GA models and hence fully subsuming existing different algebras (linear algebra, (dual) quaternion) under a single representation in a GA algebraic framework, inlcuding translation and scaling transformations. We thus aim to compare GA rotors of different models (e.g. homogeneous GA model) with respect to the basic Euclidean vector model in order to perform transformations and specifically the ones that have been problematic in skinned characters (e.g. candy-wrapper effect as a result of extreme rotations in specific joints, vertex collapsing etc.). Also as all algebras are specified and executed in CPU a GPU implementation of parts of the GA algebra would also be interesting to pursue.

In this manner we believe that GA is a forward-looking integrated algebraic framework not only limited to transformations for character animation and could be also be applied in real-time character rendering (e.g. rotation of spherical harmonics based on GA rotors for Precomputed Radiance Transfer rendering algorithms).

References

CHAUDHURI, P., PAPAGIANNAKIS, G., AND MAGNENAT-THALMANN, N. 2008. Self Adaptive Animation based on User Perspective. *The Visual Computer, Springer-Verlag 24*, 7-9, 525–533.

DORST, L., FONTIJNE, D., AND MANN, S. 2007. Geometric Algebra for Computer Science: an Object Oriented Approach to Geometry. *Morgan Kaufmann ISBN 13: 978-0-12-369465-2*, 1–620.

FONTIJNE, D. AND DORST, L. 2003. Modeling 3D euclidean geometry. *IEEE Computer Graphics and Applications*.

KAVAN, L., COLLINS, S., ZARA, J., AND CO'SULLIVAN. 2008. Geometric skinning with approximate dual quaternion blending. *ACM Transactions on Graphics*.

MAGNENAT-THALMANN, N., CORDIER, F., SEO, H., AND PAPAGIANNAKIS, G. 2004. Modeling of bodies and clothes for virtual environments. Proc. of Cyberworlds 2004, IEEE Computer Society, Tokyo, pp. 201 – 208.

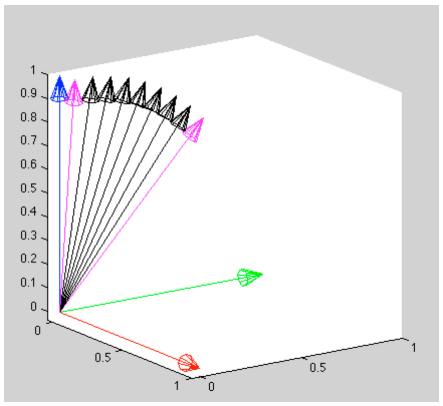
MCCARTHY, J. 1990. Introduction to theoretical kinematics. *MIT Press Cambridge, MA,* USA.

WAREHAM, R. AND LASENBY, J. 2008. Mesh Vertex Pose and Position Interpolation Using Geometric Algebra. *Springer-Verlag, Lecture Notes in Computer Science*.

APPENDIX A

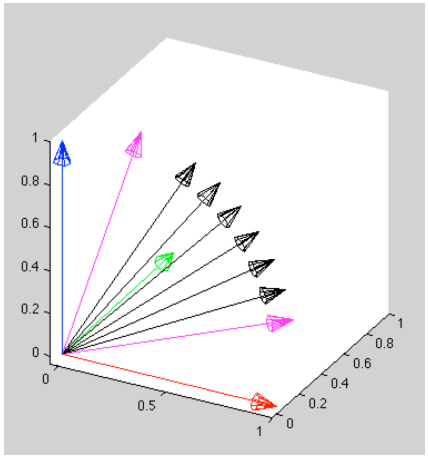
Simple 3D Vector orientation interpolation for skinning and animation using quaternions expressed as GA rotors:

```
clear
clc
clf
% src, dest quaternion interpolation by a factor
srcQ = quaternion(0,0,1,1)
srcQ = srcQ.normalize
[angleSrc, axisSrc] = AngleAxis(srcQ)
destQ = quaternion(0.0, 0.3,0.1,4.0);
destQ = destQ.normalize
[angleDest, axisDest] = AngleAxis(destQ)
factor = 1.0
quatI = slerp(srcQ, destQ, factor)

% the corresponding code in GABLE
%point interpolation experiment: P(0,1,1) rot Y(e2), 90 --> P'(1,1,0)
clf
%quat src, dest interp
draw(e1,'r');draw(e2,'g');draw(e3,'b')
srcRotor = srcQ.e(1)+srcQ.e(2)*e1+srcQ.e(3)*e2+srcQ.e(4)*e3
destRotor = destQ.e(1)+destQ.e(2)*e1+destQ.e(3)*e2+destQ.e(4)*e3

% first method of rotor calculation
u = axisSrc(1)*e1+axisSrc(2)*e2+axisSrc(3)*e3
Rsrc = gexp(-I3*u*angleSrc/2)
v = axisDest(1)*e1+ axisDest(2)*e2+ axisDest(3)*e3
Rdest = gexp(-I3*v*angleDest/2)

%second method of rotor calculation (directly from quaternion)
%Rsrc = srcQ.e(1) + srcQ.e(2)*(e2^e3) - srcQ.e(3)*(e1^e3) + srcQ.e(4)*(e1^e2)
%Rdest = destQ.e(1) + destQ.e(2)*(e2^e3) - destQ.e(3)*(e1^e3) + destQ.e(4)*(e1^e2)

Rtot = Rdest/Rsrc

for i=0 : 0.125 : factor
    Ri = gexp((i/2) *sLog(Rtot))
    Rint = (Ri * srcRotor / Ri)
    draw(Rint, 'k')
end

draw(srcRotor,'m');draw(destRotor,'m');
```



Vertex interpolation example using quaternions expressed as GA rotors:

```
clear
clc
clf
p = quaternion(0,0,1,1)% point P above [3.14, 0.7071, 0.7071, 0]
p = p.normalize
q = quaternion.angleaxis(pi/2,[0,1,0]) %rot by axis v=j (Y axis) by 90 degrees
[angle,axis] = AngleAxis(q) % retrieve angle and axis: 1.5708, [0,1,0]
q1 = conj(q) %q-1
p2 = q * p * q1 % P' new point, result is : i+j, thus point is P'(1,1,0)
qInt = slerp(p,p2,0.5) % (0.0) + i(0.40825) + j(.8165) + k(0.40825)

%corresponding code in Geometric Algebra
%point interpolation experiment: P(0,1,1) rot Y(e2), 90 --> P'(1,1,0)
clf
draw(e1,'r');draw(e2,'g');draw(e3,'b')
P=unit(e2+e3)
R=gexp(-I3*e2*pi/2/2)
Rp=R*P/R

% rotor interpolation between two interpolated points
n=8
Rtot=Rp/P
Rstep = gexp(sLog(Rtot)/n)
Rint = Rstep*P/Rstep

for i=1:n-1
    draw(Rint, 'black')
    Rint = Rstep * Rint
end

draw(P,'m')
draw(Rp,'m')
```



APPENDIX B

C++ code to convert quaternion to GA rotor interpolation (based on open-source libraries GLM for quaternion algebra, libGASandbox for GA and ASSIMP for virtual character asset loading, animation and skinning)

```
#define GLM_SWIZZLE
#define GLM_FORCE_INLINE
#include <glm/glm.hpp>
#include <glm/gtx/string_cast.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/quaternion.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <glm/gtc/random.hpp>
#include <glm/gtc/constants.hpp>

#define USE_LIBGASANDBOX
#ifdef USE_LIBGASANDBOX
#include <libgasandbox/e3ga.h>
#include <libgasandbox/e3ga_util.h>
#endif

void interpolateGA(aiQuaternion& Out, const aiQuaternion& StartRotationQ, const aiQuaternion&
EndRotationQ, float Factor)
{
    glm::quat srcQ(StartRotationQ.w,StartRotationQ.x,StartRotationQ.y,StartRotationQ.z);
    srcQ = glm::normalize(srcQ);
    float angleSrc;
    glm::vec3 axisSrc;
    angleSrc = glm::angle(srcQ);
    axisSrc = glm::axis(srcQ);

    glm::quat destQ(EndRotationQ.w, EndRotationQ.x,EndRotationQ.y,EndRotationQ.z);
    destQ = glm::normalize(destQ);
    float angleDest;
    glm::vec3 axisDest;
    angleDest = glm::angle(destQ);
    axisDest = glm::axis(destQ);

    e3ga::mv srcMV = e3ga::unit_e(srcQ.w + (srcQ.x*e3ga::e1)+ (srcQ.y*e3ga::e2) + (srcQ.z*e3ga::e3) );
    e3ga::mv dstMV = e3ga::unit_e(destQ.w + (destQ.x*e3ga::e1)+ (destQ.y*e3ga::e2)+ (destQ.z*e3ga::e3) );

    e3ga::mv u = e3ga::unit_e(axisSrc.x*e3ga::e1+axisSrc.y*e3ga::e2+axisSrc.z*e3ga::e3);
    e3ga::rotor Rsrc = _rotor( e3ga::exp( _bivector(glm::radians(angleSrc)/2 * (-e3ga::I3 * (u) )) ));
    // second alternative method for Quaternion -> GA rotor
    //e3ga::rotor Rsrc(e3ga::rotor_scalar_e1e2_e2e3_e3e1, srcQ.w,srcQ.z,srcQ.x,-srcQ.y);

    e3ga::mv v = e3ga::unit_e(axisDest.x*e3ga::e1+axisDest.y*e3ga::e2+axisDest.z*e3ga::e3);
    e3ga::rotor Rdest = _rotor( e3ga::exp( _bivector(glm::radians(angleDest)/2 * (-e3ga::I3 * (v) ))) );
    // second alternative Rotor method (remove comments below to use it and comment above, previous line)
    //e3ga::rotor Rdest(e3ga::rotor_scalar_e1e2_e2e3_e3e1, destQ.w,destQ.z,destQ.x,-destQ.y);

    //here we calculate the rotation interpolation
    e3ga::mv RtotQ = Rdest * e3ga::inverse(Rsrc);
    e3ga::mv RiQ = _rotor( e3ga::exp( _bivector(Factor/2 * e3ga::log(_rotor(RtotQ)) )) );
    e3ga::mv RintQ = RiQ * srcMV * e3ga::inverse(RiQ);

    Out.w = e3ga::_float(RintQ);
    Out.x = RintQ.e1();
    Out.y = RintQ.e2();
    Out.z = RintQ.e3();
}
```