Modeling 3D Euclidean Geometry

Daniel Fontijne and Leo Dorst
University of Amsterdam

Three-dimensional Euclidean geometry can be modeled in several ways. We compare the elegance and performance of five such models in a ray-tracing application.

The space we live in is well described as 3D Euclidean geometry for most computer graphics applications. Although it would seem straightforward to directly implement this for realistic-image generation and object simulation including their properties, most computer graphics programmers find a more indirect method attractive. That is, we prefer constructing a computational model of the 3D geometry to implement. This often improves programs in structure and efficiency. For example, the widespread use of homogeneous coordinates, which uses a 4D linear algebra to perform some of the 3D Euclidean geometry. But the vectors from 3D linear algebra also have their uses, as do quaternions—which appear to live in a 4D complex algebra—and even Plücker coordinates—which describe 3D lines using an unfamiliar 6D space.

In fact, the choice of models is getting confusing. Explanatory papers often suggest different algebras for different aspects of geometry.[1-4] Our programs typically reflect this approach. To many, the recently discovered geometric algebra appears to be just one more possibility, but there is another way of looking at this scheme.[5-7] Instead, in geometric algebra we finally have a framework containing all these modeling options and approaches in an organized manner. This approach streamlines applications by assigning various tricks such as quaternions and Plücker coordinates to a proper geometric algebra of appropriate real, interpretable vector spaces.

This article compares five models of 3D Euclidean geometry—not theoretically, but by demonstrating how to implement a simple recursive ray tracer in each of them. It's meant as a tangible case study of the profitability of choosing an appropriate model, discussing the trade-offs between elegance and performance for this particular application. This article acts as a practical sequel to two tutorials on geometric algebra previously published in *IEEE CG&A*, and we frequently reference those tutorials.[5-6]

The models we compare are

- 3D linear algebra (3D LA);
- 3D geometric algebra (3D GA), which naturally absorbs the quaternions into 3D real geometry;
- 4D linear algebra (4D LA), that is, the familiar homogeneous coordinates;
- 4D geometric algebra (4D GA), implements the homogeneous model, which naturally absorbs Plücker coordinates of lines and planes into homogeneous computations; and
- 5D geometric algebra (5D GA), which implements the conformal model. This model gives coordinates to circles and spheres and provides the most compact expressions for 3D Euclidean computations known to date.

We picked both 3D LA and 4D LA because we wanted a basic and an advanced mainstream model as a baseline. We selected 3D GA and 4D GA because they are the (improved) GA variants of the 3D LA and 4D LA models. The 5D GA model demonstrates the possible improvements when using more sophisticated models. Although we don't explicitly use Grassmann spaces as recommended by Goldman,[2] we shall show that using geometric algebra to implement Grassmann spaces significantly extends their applicability.

We choose a ray tracer as a benchmark for the following reasons:

- Everyone familiar with computer graphics knows how a basic ray tracer works and possibly has implemented one.
- Implementing the core of a ray tracer is possible with a relatively small amount of code. This was important because we had to write many different implementations of the same algorithm.
- A ray tracer contains a diverse selection of geometric computations, such as rotation, translation, reflection, refraction, (signed) distance computation, and

3D LA
3D GA
4D LA
4D GA
5D GA

1 **These images are identical, but we rendered each using a different 3D Euclidean geometry model. The scene consists of five objects—a transparent drinking glass, reflective sphere, red diffuse sphere, and textured/bump mapped wood piece—modeled with 7,800 triangles.**

line-plane and line-sphere intersection. This lets us to show by example how to perform these computations in different models.

But we emphasize that our main goal here is to compare frameworks for representation and computation of geometry in some practical situation, not to build a ray tracer per se. The resulting ray tracer is not a marvel of contemporary computer graphics; yet it's sufficiently sophisticated to render images such as those shown in Figure 1.

Ray tracer

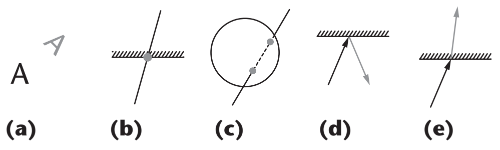
We use a basic recursive ray-tracing algorithm, without special techniques for efficiency, except for the use of a binary-space-partitioning (BSP) tree to accelerate ray-mesh intersection computations. Describing the precise algorithm in great detail is not meaningful here; only the geometric computations matter to the discussion at hand. A more detailed specification is available elsewhere (http://www.science.uva.nl/~fontijne/raytracer).

The ray-tracing algorithm accepts as input a description of the scene, including camera, lighting, and polygonal model information such as position, orientation, shape; and material properties. For each image pixel, a ray is traced through the scene as it hits models and possibly gets reflected and refracted. Where a ray hits a surface, we perform lighting computations for each visible or ambient light source. The weighted sum of such lighting computations determines the final color of each pixel.

The ray-tracing algorithm requires representations of geometric primitives such as vectors, points, lines, planes, spheres, as well as transformations of these primitives. In some models, primitives can also act as operators. For instance, using geometric algebra, a plane can be applied to another primitive directly to reflect that primitive in the plane.

The geometric computations and operations that we must implement in each model to build our ray tracer are

- rotation and translation of arbitrary primitives (points, lines, planes);
- reflection and refraction (Snell's law) of directed lines;
- test for and computation of the intersection of lines and planes, lines and triangles, and lines and spheres;
- computation of the angle between lines and/or the angle between planes; and
- computation of the distance between two points and the signed distance of points to planes.



2 **Illustration of the geometric computations that we treat in detail for each model. Computation input is shown in black; output in gray. (a) Translation and rotation. (b) Intersection of a line and a plane. (c) Intersection of a line and a sphere. (d) Reflection of a directed line in a plane. (e) Refraction of a directed line in a plane.**

We do not give a detailed specification here of each of the geometric computations we discuss in this article. Writing down those descriptions implies the use of a specific model, because using a model is the only method we know to precisely encode geometry. An important theme of this article is that the use of any model, even a model of 3D Euclidean geometry, determines how you implement your solution and also shapes the way you think about the problem. To remain impartial with respect to the five models, we don't use one of those models at this point to specify the geometric computations. Instead, we include a graphical representation in Figure 2. The icons in this figure show only the relevant geometric primitives. Derived geometric primitives such as angles, intermediate intersection points, and surface normals arise from the manner that we implement the computations in mainstream models of 3D Euclidean geometry and don't necessarily arise in other models. For example, when we treat the model, a directed line can be reflected in a plane without using a surface normal or the intersection point of the line and the plane.

Models

Our introductions to the five models show only how they represent some important primitives and rotation/translation operations. For the novel GA models, we give references to sources that provide more detail. After each introduction, we show the equations that implement the five geometric computations from Figure 2. Readers with little mathematical background shouldn't be discouraged by these equations. Instead we encourage all readers not to focus on understanding the equations, but to read with a bird's eye view and skip back and forth between the five sections to compare the

Mathematical Notations

We employ the following notation across all models: lowercase Greek symbols (ρ, δ, ϕ) denote scalars. Lowercase bold symbols ($\mathbf{u}$, $\mathbf{q}$, $\mathbf{s}$) represent elements of the algebra interpreted as geometric primitives (directions, points, spheres). Uppercase bold symbols ($\mathbf{R}$, $\mathbf{M}$) denote elements of the algebra interpreted as operators (rotors, transformation matrices). Lowercase plain symbols with an arrow overhead ($\vec{v}$, $\vec{u}$) occasionally denote vectors that aren't strictly elements of the algebra at hand. When possible, equations appear close to the form in which they are implemented in actual code.

length and simplicity of the equations and the number of split-up cases.

Knowing that there exist alternative ways to implement each geometric operation in each model, we use the most obvious approach in each model. The equations we use to implement the geometric operations in the 3D LA model are virtually identical to those quoted in Glassner as most efficient.[8] The "Mathematical Notation" sidebar defines the notations we use in the models.

3D linear algebra

In the 3D LA model, vectors and scalars represent all primitives. A vector that points from the origin to the location of the point represents a point. A vector pointing from the origin to some point on a line and a unit vector pointing along the direction of a line represents a line. A normal vector and a scalar that gives the distance of the plane to the origin represent a plane. A vector pointing from the origin to the center of a sphere and a scalar giving the radius represents a sphere. We explicitly represent each primitive relative to a specific origin that we chose a priori. A vector represents translation. Because it's a linear mapping, a 3×3 matrix represents rotation about the origin.

We made all geometric computations using matrix-vector multiplication, addition and subtraction of vectors, scalar multiplication, dot products (denoted by $\bullet$), and cross products (denoted by $\times$).

Rotation/translation. A point $\mathbf{q}$ is rotated/translated as $\mathbf{q}' = \mathbf{R}\mathbf{q} + \mathbf{t}$, where $\mathbf{R}$ is a rotation matrix, and $\mathbf{t}$ is a translation vector. To translate/rotate a line (given by point $\mathbf{q}_l$ on the line and a unit vector $\mathbf{u}$ along the line), we compute $\mathbf{q}'_l = \mathbf{R}\mathbf{q}_l + \mathbf{t}$ and $\mathbf{u}' = \mathbf{R}\mathbf{u}$. A plane (given by its unit normal vector $\mathbf{n}$ and the scalar distance to the origin δ) is rotated/translated by $\mathbf{n}' = \mathbf{R}\mathbf{n}$ and $\delta' = \delta + \mathbf{t} \bullet \mathbf{n}'$ (http://carol.wins.uva.nl/~fontijne/raytracer/files/raytracer_primitives.pdf and http://carol.wins.uva.nl/~fontijne/raytracer/files/raytracer_operations.pdf)

Line-plane intersection. We can compute the intersection point $\mathbf{q}_i$ of a line and a plane as

$$\mathbf{q}_i = \mathbf{q}_l - \frac{((\mathbf{q}_l \bullet \mathbf{n}) - \delta)\mathbf{u}}{\mathbf{u} \bullet \mathbf{n}} \quad (1)$$

if $\mathbf{u} \bullet \mathbf{n}$ is not equal to 0.

Line-sphere intersection. We can compute the two intersection points $\mathbf{q}_-$ and $\mathbf{q}_+$ of a line and a sphere (given by its center point $\mathbf{q}_s$ and its scalar radius ρ). First, the closest point $\mathbf{q}_c$ to the center of the sphere on the line is computed as $\mathbf{q}_c = \mathbf{q}_l + ((\mathbf{q}_s - \mathbf{q}_l) \bullet \mathbf{u})\mathbf{u}$, then the normalized squared Euclidean distance, δ_n^2 of $\mathbf{q}_c$ to $\mathbf{q}_s$ determines if the line intersects the sphere:

$$\delta_n^2 = \frac{(\mathbf{q}_c - \mathbf{q}_s) \bullet (\mathbf{q}_c - \mathbf{q}_s)}{\rho^2}$$

If δ_n^2 is larger than 1, the line and the sphere do not intersect. If δ_n^2 is exactly 1, $\mathbf{q}_c$ is the single intersection point. Otherwise $\mathbf{q}_-$ and $\mathbf{q}_+$ can be computed as

$$\mathbf{q}_\pm = \mathbf{q}_c \pm \rho\sqrt{1 - \delta_n^2}\,\mathbf{u}$$

Reflection. The reflected direction $\mathbf{u}'$ of a line in a plane, can be computed as

$$\mathbf{u}' = -2(\mathbf{n} \bullet \mathbf{u})\mathbf{n} + \mathbf{u} \quad (2)$$

The reflected line would then be given by $\mathbf{q}_i$ (the intersection point of the line and the plane) and $\mathbf{u}'$. Note that we have to explicitly compute $\mathbf{q}_i$ (using Equation 1) before we obtain a full representation of the reflected line.

Snell's law. As a ray travels from one medium to another, its direction gets refracted according to Snell's law:

$$\frac{\sin\phi_1}{\sin\phi_2} = \frac{\eta_2}{\eta_1}$$

where ϕ_1 is the incoming angle, ϕ_2 is the outgoing angle, and η_1 and η_2 are the refractive indices of the media. In the "Derivation of the 3D Geometric Algebra Refraction Equation" sidebar we use geometric algebra to compactly derive the classical equation for implementing Snell's law. Here we only present the result of that derivation, translated to 3D LA.

The unit surface normal of the (tangent-) plane separating the media is given by $\mathbf{n}$. The unit direction of the line is given by $\mathbf{u}$. We define $\eta = \eta_2/\eta_1$, the refractive index of medium 2 relative to medium 1. This is all we need to compute the refracted direction of the line:

$$\mathbf{u}' = \left(\operatorname{sign}(\mathbf{n}\bullet\mathbf{u})\sqrt{1 - \eta^2 + (\mathbf{n}\bullet\mathbf{u})^2\eta^2} - (\mathbf{n}\bullet\mathbf{u})\eta\right)\mathbf{n} + \eta\mathbf{u} \quad (3)$$

3D geometric algebra

Three-dimensional geometric algebra is an extension of 3D linear algebra.[5,6] It has an operation to span subspaces through the origin: the outer product[5] denoted by the $\wedge$ symbol. Such subspaces or blades[5] are the basic elements of computation. In 3D GA, we interpret the bivectors, or 2-blades, (of the form $\mathbf{a} \wedge \mathbf{b}$) as oriented, directed planes through the origin. We use bivectors instead of normal vectors because they encode the same information but behave better under linear transformations. We can naturally extend the inner product

Derivation of the 3D Geometric Algebra Refraction Equation

We use 3D geometric algebra (3D GA) to derive 3D linear algebra (3D LA) Equation 3 (in the main article text), which we repeat here:

$$\mathbf{u}' = \left(\operatorname{sign}(\mathbf{n} \bullet \mathbf{u})\sqrt{1-\eta^2+(\mathbf{n}\bullet\mathbf{u})^2\eta^2} - (\mathbf{n}\bullet\mathbf{u})\eta\right)\mathbf{n} + \eta\mathbf{u}$$

You might compare this with work found in Glassner,[1] which contains two 3D LA derivations of the same equation. In these equations, **u** is the direction of the incoming ray; **n** is the dual of the bivector **p** representing the plane, that is, the normal vector; and $\eta = \eta_1/\eta_2$ is a constant depending on the speed of light in both media. We compute **u′**, the direction of the outgoing ray. In 3D GA, Snell's law can be fully specified by this set of equations:

$$\mathbf{u}' \wedge \mathbf{n} = \eta\mathbf{u} \wedge \mathbf{n} \qquad \text{(A)}$$
$$\mathbf{u}'^2 = \mathbf{u}^2 \qquad \text{(B)}$$
$$\operatorname{sign}(\mathbf{u}' \bullet \mathbf{n}) = \operatorname{sign}(\mathbf{u} \bullet \mathbf{n}) \qquad \text{(C)}$$

Equation A states that **u**, **u′**, and **n** must all lie in the same plane, while the sizes of both bivectors are related by the constant η. Equation B simply states that the lengths of **u′** and **u** must be equal, while Equation C states that **u′** and **u** must both have the same heading with respect to **n**. We will extract **u′** from Equation A. The sum of the inner and outer product of **u′** and **n** is equal to their geometric product

$$\mathbf{u}'\mathbf{n} = \mathbf{u}' \bullet \mathbf{n} + \mathbf{u}' \wedge \mathbf{n} \qquad \text{(D)}$$

This suggests that, if we could express **u′ • n** without using **u′**, we could get the answer by adding **u′ • n** to Equation A's left hand side and dividing by **n**. To find an expression for **u′ • n**, we note that

$$\begin{aligned}\mathbf{n}^2\mathbf{u}^2 &= \mathbf{n}^2\mathbf{u}'^2 = \mathbf{n}\mathbf{u}'\mathbf{u}'\mathbf{n}\\ &= (\mathbf{n} \bullet \mathbf{u}' + \eta(\mathbf{n} \wedge \mathbf{u}))(\mathbf{u}' \bullet \mathbf{n} + \eta\,(\mathbf{u} \wedge \mathbf{n}))\\ &= (\mathbf{u}' \bullet \mathbf{n})^2 - \eta(\mathbf{u}' \bullet \mathbf{u})(\mathbf{u} \wedge \mathbf{n}) + \eta(\mathbf{u}' \bullet \mathbf{u})\,(\mathbf{u} \wedge \mathbf{n}) - \eta^2(\mathbf{u} \wedge \mathbf{n})^2\\ &= (\mathbf{u}' \bullet \mathbf{n})^2 - \eta^2(\mathbf{u} \wedge \mathbf{n})^2\end{aligned}$$

From this and Equation C it follows that

$$\mathbf{u}' \bullet \mathbf{n} = \operatorname{sign}(\mathbf{u} \cdot \mathbf{n})\sqrt{\mathbf{n}^2\mathbf{u}^2 + \eta^2(\mathbf{u} \wedge \mathbf{n})^2} \qquad \text{(E)}$$

If we now add Equation E to Equation A we get:

$$\mathbf{u}' \wedge \mathbf{n} + \mathbf{u}' \bullet \mathbf{n} = \eta\mathbf{u} \wedge \mathbf{n} + \operatorname{sign}\,(\mathbf{u} \bullet \mathbf{n})\sqrt{\mathbf{n}^2\mathbf{u}^2 + \eta^2(\mathbf{u} \wedge \mathbf{n})^2} \qquad \text{(F)}$$

If we compare the left hand side of Equation F to the right hand side of Equation D, we see that Equation F is the (invertible) geometric product of **u′** and **n**, so we divide by **n** to finish:

$$\mathbf{u}' = \frac{\eta\mathbf{u} \wedge \mathbf{n} + \operatorname{sign}(\mathbf{u} \bullet \mathbf{n})\sqrt{\mathbf{n}^2\mathbf{u}^2 + \eta^2(\mathbf{u} \wedge \mathbf{n})^2}}{\mathbf{n}}$$

If both **n** and **u** have unit length we can simplify this to

$$\mathbf{u}' = \eta(\mathbf{u} \wedge \mathbf{n})\mathbf{n} + \left(\operatorname{sign}(\mathbf{u} \bullet \mathbf{n})\sqrt{1+\eta^2(\mathbf{u} \wedge \mathbf{n})^2}\right)\mathbf{n}$$

$$= \eta\mathbf{u} + \left(\operatorname{sign}(\mathbf{u} \bullet \mathbf{n})\sqrt{1+\eta^2(\mathbf{u} \wedge \mathbf{n})^2} - \eta\mathbf{u} \bullet \mathbf{n}\right)\mathbf{n}$$

The last step is apply the fact

$$(\mathbf{u} \wedge \mathbf{n})^2 = (\mathbf{u} \bullet \mathbf{n})^2 - 1$$

This is the geometric algebra equivalent of $-\sin^2\phi = \cos^2\phi - 1$.

Reference

1. A.S. Glassner, ed., *An Introduction to Ray Tracing*, Academic Press, 1989.

(denoted by the • symbol) to blades, and this is useful for projection and metric relationships.

GA also has a geometric product,[5] denoted by a half space symbol, as in **a b**. The geometric product permits multiplication and division[5] by vectors and subspaces. The ratio of two vectors form a rotor,[6] which we can use as a rotation operator. In fact, the rotor has the same properties as a quaternion, but within the context of geometric algebra, it's a real operator that can rotate subspaces of any grade. Alternatively, we can construct a rotor as the exponential of the bivector representing the rotation plane and angle.

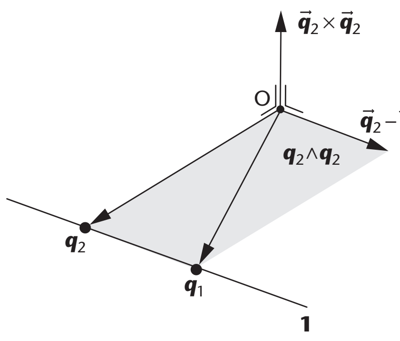
Besides the various products, we also use addition, subtraction, and inversion. The dual operator,[6] denoted by a superscript *, returns the dual of any blade, that is, the orthogonal complement in 3D space. These constructions naturally extend to *n*-dimensional vector spaces.

Eight coordinates relative to an 8D basis can represent a general number or multivector in 3D GA: one for a scalar component, three for vector components, three for bivector components, and one for a trivector component (3-blades, interpreted as volume elements).

Rotation/translation. We perform rotation of a vector about an axis through the origin in 3D GA with $\mathbf{v}' = \mathbf{R}\,\mathbf{v}\,\mathbf{R}^{-1}$. The vector is sandwiched between the rotor **R** and its inverse $\mathbf{R}^{-1}$. We create **R** as $\mathbf{R} = \exp(-1/2\phi\mathbf{b}) = \cos 1/2\phi - \mathbf{b}\sin 1/2\phi$, where ϕ is the angle of rotation and **b** the unit bivector denoting the plane of rotation. Such an **R** is normalized. This implies that $\mathbf{R}^{-1}$ is equal to $\mathbf{R}^{\sim}$, the reverse of **R**.[5] We can efficiently compute the reverse by sign flipping part of the coordinates of **R**.

Sandwiching operations like $\mathbf{R}\,\mathbf{v}\,\mathbf{R}^{-1}$ are common in GA; they typically apply objects like rotors to blades. Once you replace rotation matrix multiplication by this rotor sandwiching operation, points and lines are transformed the same way in 3D GA as in 3D LA.

A plane is now given by its bivector **b** and its scalar



3 Plücker coordinates of a line in 4D LA and 4D GA. Vector $\vec{q}_1 - \vec{q}_2$ gives the lines direction. Vector $\vec{q}_1 \times \vec{q}_2$ encodes both the distance of the line to the origin and the normal of the plane through the origin in which $\mathbf{q}_1$ and $\mathbf{q}_2$ lie. In 4D GA, the single bivector $\mathbf{q}_1 \wedge \mathbf{q}_2$ (illustrated by the shaded area) describes the entire line.

distance to the origin δ, and is rotated/translated as follows: $\mathbf{b}' = \mathbf{R}\,\mathbf{b}\,\mathbf{R}^{-1}$ and $\delta' = \delta + (\mathbf{t} \wedge \mathbf{b}')^*$. Here, $(\mathbf{t} \wedge \mathbf{b}')^*$ is equal to $\mathbf{t} \bullet (\mathbf{b}'^*)$, but is slightly more efficient.

Line-plane intersection. The intersection $\mathbf{q}_i$ point of a line and a plane can be computed as

$$\mathbf{q}_i = \mathbf{q}_l - \frac{((\mathbf{q}_l \wedge \mathbf{b})^* - \delta)\mathbf{u}}{(\mathbf{u} \wedge \mathbf{b})^*}$$

if $\mathbf{u} \wedge \mathbf{b}$ is not equal to 0.

Line-sphere intersection. We handle line-sphere intersection in the same way in 3D GA as in 3D LA, except we replace the dot products by equivalent inner products.

Reflection. The reflected direction $\mathbf{u}'$ of a line in a plane can be computed as

$$\mathbf{u}' = -\,\mathbf{b}\,\mathbf{u}\,\mathbf{b}^{-1} = \mathbf{b}\,\mathbf{u}\,\mathbf{b} \qquad (4)$$

As with rotation, we see that $\mathbf{u}$ is sandwiched between the two $\mathbf{b}$'s. The reflected line would be given by $\mathbf{q}_i$ and $\mathbf{u}'$. We must explicitly compute $\mathbf{q}_i$ before we obtain a full representation of the reflected line.

Snell's law

The 3D GA model implements Snell's law in the same way as the 3D LA model. In the 3D GA model, we only have to set $\mathbf{n} = \mathbf{b}^*$, and implement the rest as in 3D LA Equation 3.

4D linear algebra

This is the most incoherent of all the models presented in this article, although it's probably representative of what an advanced computer graphics programmer would use. The model uses homogeneous coordinates, Plücker coordinates, and 4 × 4 transformation matrices to implement part of an oriented projective geometry, such as described in Stolfi.[9]

The use of homogeneous coordinates provides an extra basis vector or axis, besides the standard *x*-, *y*-, and *z*-axes. The extra coordinate required for the new basis vector is usually called **w**. The fourth dimension is used so that we can represent arbitrary affine subspaces (that is, lines and planes floating in space) as elements of direct computation. It also lets us encode all affine transformations on points and vectors in the well known 4 × 4 transformation matrices. The 4D homogeneous coordinates of a point **q** are a 3-vector $\vec{q}$ that points from the origin to the point, plus one extra coordinate, set to 1, that refers to the *w*-axis. We can thus denote a point as $\mathbf{q} = (\vec{q} : 1)$. The 4D homogeneous coordinates of an ordinary 3D vector are $\mathbf{v} = (\vec{v} : 0)$. The 4-vectors $\mathbf{q} = (\alpha\vec{q} : \alpha)$, where α is not 0, can be safely interpreted and used as points by introducing normalization $\mathbf{q}_n = (\vec{q} : 1)$. This is also the natural place to start applying the Grassmann space interpretation found in Goldman.[2]

Plücker coordinates are the homogeneous coordinates of lines and planes and are useful for intersection and relative orientation computations. They are natural in the 4D GA context. Classically, they are rarely introduced geometrically as a natural extension of homogeneous coordinates. Perhaps this is why they are not often used and are underappreciated.

The Plücker coordinates of a line **l** through two points $\mathbf{q}_1 = (\vec{q}_1 : 1)$ and $\mathbf{q}_2 = (\vec{q}_2 : 1)$ are $\mathbf{l} = (\vec{q}_1 - \vec{q}_2 : \vec{q}_1 \times \vec{q}_2) = (\vec{q}_1 - \vec{q}_2 : (\vec{q}_1 - \vec{q}_2) \times \vec{q}_1)$. Hence, six coordinates that can be grouped into two 3-vectors represent a line, as Figure 3 illustrates.

The Plücker coordinates of a plane are the normal vector $\vec{n}$ of the plane and its scalar distance to the origin δ: $\mathbf{p} = [\vec{n} : \delta]$. (We use square brackets to distinguish between the Plücker coordinates of points and planes.)

Geometric computations in this model are made using matrix vector multiplication, addition and subtraction of various objects, scalar multiplication, 3D vector dot and cross products, and special Plücker products. To perform the Plücker products, we often must break up the coordinates into scalars and 3D vectors, perform some computations on them, and reassemble them into a Plücker object. When we multiply a 4 × 4 affine transformation matrix **M** with a 3-vector $\vec{v}$, this is shorthand for the $\vec{v}'$ part of $(\vec{v}' : 0) = \mathbf{M}(\vec{v} : 0)$.

Rotation/translation. A point **q** is rotated/translated through multiplying it by a 4 × 4 transformation matrix **M**, or $\mathbf{q}' = \mathbf{M}\mathbf{q}$. Such a simple product between the transformation matrix **M** and the Plücker coordinates of a line **l** does not exist. Although we could devise a new type of 6 × 6 affine transformation matrix, here, we separate the line into a point and a direction, transform them, and reconstruct the line: $\mathbf{l} = (\vec{u} : \vec{v})$, $\mathbf{q}' = (\vec{q}' : 1) = \mathbf{M}(\vec{v} \times \vec{u} : 1)$, and $\mathbf{l}' = (\mathbf{M}\vec{u} : \mathbf{M}\,\vec{u} \times \vec{q}')$.

We can't directly multiply a plane **p** with a transformation matrix **M**. But if **M** contains only rotations and translations, then we can derive that $\mathbf{p}' = \mathbf{M}^{-T}\mathbf{p}$ is the transformed plane. ($\mathbf{M}^{-T}$ is the inverse of the transpose of **M**).

Line-plane intersection. Here, we demonstrate the first valuable use of Plücker coordinates in our ray tracer. The intersection point **q** of a line $\mathbf{l} = (\vec{u} : \vec{v})$ and a plane $\mathbf{p} = [\vec{n} : \delta]$ is

$$\mathbf{q} = \left(\frac{\vec{v} \times \vec{n} + \delta\vec{u}}{\vec{u} \bullet \vec{n}} : 1\right) \tag{5}$$

In practice, we implement equations like this using the Plücker coordinates directly, without explicitly constructing the vectors $\vec{n}, \vec{u}$, and $\vec{v}$. For this purpose, special multiplication tables are available.[9]

Line-sphere intersection. To compute the two intersection points $\mathbf{q}_-$ and $\mathbf{q}_+$ of a line $\mathbf{l} = (\vec{u} : \vec{v})$ and a sphere (given by its center point $\mathbf{q}_s = (\vec{q}_s : 1)$ and its scalar radius ρ), we proceed as in 3D linear algebra. Only the computation of the point $\mathbf{q}_c$ on the line closest to the center of the sphere is performed differently. First, we translate **l** over the vector $\vec{t} = -\vec{q}_s$ such that the center of the sphere is at the origin. Then, we can compute the point on **l** closest to the center of the sphere:

$$\mathbf{l}_t = (\vec{u}_t : \vec{v}_t) = (\vec{u} : \vec{v} + \vec{u} \times \vec{t})$$

$$\mathbf{q}_c = \left(\frac{\vec{v}_t \times \vec{u}_t}{|\vec{u}_t|} : 1\right)$$

The rest of the computations are analogous to those in the 3D LA model.

Reflection. To reflect a line $\mathbf{l} = (\vec{u} : \vec{v})$ in a plane $\mathbf{p} = [\vec{n} : \delta]$, we first reflect the direction $\vec{u}$ of the line $\vec{u}' = -2(\vec{n} \bullet \vec{u})\vec{n} + \vec{u}$ and then construct a new line from the intersection point **q** of **l** and **p**, and the reflected direction $\vec{u}'$. We have to explicitly compute **q** (using Equation 5) before we obtain a full representation of the reflected line.

Snell's law. Snell's law is handled using the same technique we used to reflect a line. We take the line apart in an intersection point and a direction, then compute everything as we did in 3D LA, and construct a new line from those results.

4D geometric algebra

Here, we revise the 4D LA section using GA. We call this the 4D homogeneous *model* as opposed to 4D homogeneous *coordinates* to denote that it naturally encompasses all geometric elements, not just points. Mann and Dorst give more details on the model.[6]

Again we will use an extra basis vector representing the point at the origin. But, following convention, we call it $\mathbf{e}_0$ instead of **w**. As with any Euclidean unit vector, $\mathbf{e}_0 \bullet \mathbf{e}_0 = 1$. The *x*-, *y*-, and *z*-axes are called $\mathbf{e}_1$, $\mathbf{e}_2$, and $\mathbf{e}_3$. Points are defined as $\mathbf{q} = \vec{q} + \mathbf{e}_0$, where $\vec{q}$ is a Euclidean 3D vector that points from the origin to **q**. Therefore, in the model vectors with an $\mathbf{e}_0$ component of zero represent 3D vectors by themselves. We can add 3D vectors to points to produce translated points.

To construct a line **l** from two points $\mathbf{q}_1$ and $\mathbf{q}_2$, we wedge them together forming a bivector:

$$\mathbf{l} = \mathbf{q}_1 \wedge \mathbf{q}_2 \tag{6}$$

If we choose the appropriate bivector basis for our 4D GA, the six coordinates of **l** are exactly the Plücker coordinates of the line. Figure 3 illustrates this.We construct a plane **p** by wedging three points together: $\mathbf{p} = \mathbf{q}_1 \wedge \mathbf{q}_2 \wedge \mathbf{q}_3$. Again, with the appropriate trivector basis, the four coordinates of trivector **p** are identical to the Plücker coordinates of the plane.We often use $\mathbf{e}_0 \bullet \mathbf{l}$ and $\mathbf{e}_0 \bullet \mathbf{p}$ to retrieve the direction elements of a line or a plane, resulting in a purely Euclidean vector or bivector.We can naturally make a linear transformation *f* (such as rotation, translation, and projection) on vectors act on blades (lines and planes) by demanding $f(\mathbf{a} \wedge \mathbf{b}) = f(\mathbf{a}) \wedge f(\mathbf{b})$ for all vectors **a** and **b**. This is called an outermorphism. If a transformation is an outermorphism, we can construct for it an outermorphism operator. The outermorphism operator is the matrix representation of the linear transformation. We can use it to transform any primitive (vector, point, line, or plane). A 4×4 matrix transforms points, a 6×6 matrix transforms lines, and another 4×4 matrix transforms planes. The 4×4 matrix that transforms points and vectors is exactly the traditional 4×4 transformation matrix used in homogeneous coordinates. We can naturally construct the outermorphism operator in GA by applying the preceding definition. To carry out our geometric computations in this section, we use the standard set of GA products and operators as with 3D GA, plus the outermorphism operator.

Rotation/translation. We can rotate/translate any primitive **x** by applying outermorphism operator **M**: $\mathbf{x}' = \mathbf{M}\mathbf{x}$. It's not necessary to split this operation into different cases (point, vector, line, or plane) as in the 4D LA model. Outermorphisms automatically handle each case correctly.

Line-plane intersection. The intersection point **q** of a line **l** and a plane **p** is given by $\mathbf{q} = \mathbf{p}^* \bullet \mathbf{l}$. This is the standard primitive intersection equation in the model. For example, we can also use the equation to compute the intersection of two planes, or even of two lines, given that we compute dual (*) with respect to the correct (sub-) space, as detailed in Mann and Dorst.[6] In general the point **q** will not be normalized; we can enforce this by dividing **q** by $\mathbf{e}_0 \bullet \mathbf{q}$.

Line-sphere intersection. As in the 4D LA model, we first compute the closest point $\mathbf{q}_c$ on the line **l** to the sphere (given by its center point $\mathbf{q}_s$ and its radius ρ). We translate **l** over a vector $\mathbf{t} = \mathbf{q}_s - \mathbf{e}_0$ (assuming $\mathbf{q}_s$ is normalized) such that $\mathbf{q}_s$ is at the origin:

$$\mathbf{l}_t = \mathbf{l} - \mathbf{t} \wedge (\mathbf{e}_0 \bullet \mathbf{l}) \tag{7}$$

Here we use the **t** notation, (as opposed to $\vec{t}$ in the 4D LA model) because it's still a member of the algebra since it was retrieved algebraically from $\mathbf{q}_s$. We retrieve the point $\mathbf{q}_c$ by projecting the origin onto the line and translating the result back to the original frame: $\mathbf{q}_c =$

$(\mathbf{e}_0 \bullet \mathbf{l}_t)\mathbf{l}_t^{-1} + \mathbf{t}$. We can then compute the intersection points of the line and the sphere $\mathbf{q}^+$ and $\mathbf{q}^-$, as explained previously in the 3D LA section, if we set $\mathbf{u} = \mathbf{e}_0 \bullet \mathbf{l}$.

Reflection. Unfortunately, the model doesn't let us reflect an arbitrary line **l** in an arbitrary plane **p** directly in space. So we either have to convert the technique used in the section on 4D LA to 4D GA, or we can translate **l** and **p** over a vector –**t** such that their intersection point **q** is at the origin. If the intersection point is at the origin, we can compute the reflected line directly. Once we have done that, we translate the reflected line back over the vector **t**.

$$\mathbf{q} = \frac{\mathbf{p}^* \bullet \mathbf{l}}{\mathbf{e}_0 \bullet (\mathbf{p}^* \bullet \mathbf{l})}$$

$$\mathbf{t} = \mathbf{q} - \mathbf{e}_0$$

$$\mathbf{l}_t = \mathbf{l} - \mathbf{t} \wedge (\mathbf{e}_0 \bullet \mathbf{l}) \qquad (8)$$

$$\mathbf{p}_t = \mathbf{p} - \mathbf{t} \wedge (\mathbf{e}_0 \bullet \mathbf{p}) \qquad (9)$$

$$\mathbf{1}'_t = \mathbf{p}_t \mathbf{l}_t \mathbf{p}_t$$

$$\mathbf{l}' = \mathbf{l}'_t + \mathbf{t} \wedge (\mathbf{e}_0 \bullet \mathbf{l}'_t)$$

The simple equation we use to translate **l**, **p**, and $\mathbf{l}'_t$ in Equations 7, 8, and 9 works for points, lines, and planes.

Snell's law. The use of geometric algebra in the homogeneous model doesn't let us handle Snell's law more elegantly. We still must separate the incoming line **l** into its direction ($\mathbf{u} = \mathbf{e}_0 \bullet \mathbf{l}$) and its intersection point with the plane **p** (which is $\mathbf{q} = \mathbf{p}^* \bullet \mathbf{l}$), refract the direction as with 3D GA, and construct the new line.5D geometric algebraThe 5D conformal model[7] (called the *double homogeneous model* in Mann and Dorst[6]) uses two extra basis vectors, as opposed to one in the homogeneous model. Basis vector $\mathbf{e}_0$ represents the point at the origin; basis vector $\mathbf{e}_\infty$ represents the point at infinity. These two basis vectors are reciprocal null vectors, which means $\mathbf{e}_0 \bullet \mathbf{e}_0 = \mathbf{e}_\infty \bullet \mathbf{e}_\infty = 0$ and $\mathbf{e}_0 \bullet \mathbf{e}_\infty = 1$. Besides these two special basis vectors, there are three ordinary basis vectors ($\mathbf{e}_1$, $\mathbf{e}_2$, and $\mathbf{e}_3$) that are equivalent to the traditional *x*-, *y*-, and *z*-axes.

This might seem an unusual basis for a 3D Euclidean geometry model. But, if we consider the role of the extra basis vectors informally, we can motivate them to perform some extra administration of our geometric objects' properties. This lets us more easily perform many important geometric computations.

Points are constructed as $\mathbf{q} = \vec{q} + \mathbf{e}_0 - 1/2\,(\vec{q} \bullet \vec{q})\,\mathbf{e}_\infty$, where $\vec{q}$ is a Euclidean 3D vector pointing from the origin to the location of the point **q**. Once we have defined our points, we no longer need the origin $\mathbf{e}_0$ and can construct extended objects (including lines, planes, point pairs, circles, and spheres) by wedging the appropriate points together. To construct an object, we wedge together the appropriate set of characteristic primitives required to specify the object. That is, a line **l** through the points $\mathbf{q}_1$ and $\mathbf{q}_2$ is constructed as $\mathbf{l} = \mathbf{q}_1 \wedge \mathbf{q}_2 \wedge \mathbf{e}_\infty$. (The difference between this and the 4D GA model shown in Equation 6 is that here we must also wedge $\mathbf{e}_\infty$.) We can construct a plane by wedging three points plus infinity. To construct a circle **c** through three points $\mathbf{q}_1$, $\mathbf{q}_2$, and $\mathbf{q}_3$, we construct the blade $\mathbf{c} = \mathbf{q}_1 \wedge \mathbf{q}_2 \wedge \mathbf{q}_3$ (so a line is actually a circle through infinity). To construct a sphere **s** that contains the circle **c** and a fourth point $\mathbf{q}_4$, we simply wedge them together: $\mathbf{s} = \mathbf{c} \wedge \mathbf{q}_4 = \mathbf{q}_1 \wedge \mathbf{q}_2 \wedge \mathbf{q}_3 \wedge \mathbf{q}_4$. It's easy, straightforward, and general to construct these objects. Because the outer product is antisymmetric, all objects are oriented. So the circle $\mathbf{q}_1 \wedge \mathbf{q}_2 \wedge \mathbf{q}_3$ has the opposite orientation of $\mathbf{q}_1 \wedge \mathbf{q}_3 \wedge \mathbf{q}_2$, and the line $\mathbf{q}_1 \wedge \mathbf{q}_2 \wedge \mathbf{e}_\infty$ has the opposite direction of $\mathbf{q}_2 \wedge \mathbf{q}_1 \wedge \mathbf{e}_\infty$.

We can construct rotors—used to represent rotations—as the geometric product of vectors, or as exponents of bivectors. Because we have a point at infinity, $\mathbf{e}_\infty$, we can represent translations as rotations about infinity. A translator represents a translation over the vector $\vec{t}$:

$$\mathbf{T} = 1 + \frac{1}{2}\vec{t} \wedge \mathbf{e}_\infty = \mathbf{e}^{\frac{1}{2}\vec{t}\infty}$$

This unites translations and rotations into a single versor representation. Therefore, to first apply a translation represented by **T**, followed by a rotation represented by **R**, we compute the geometric product $\mathbf{V} = \mathbf{RT}$. We can then apply this **V** to any object. This differs from the 4D LA and 4D GA models, where we can only unify translations and rotations by using transformation matrices or the outermorphism operator.

Rotation/translation. As explained previously, we can represent a sequence of rotations and translations by a single versor. We can translate and rotate any primitive **x** at the same time by applying the appropriate versor: $\mathbf{V}$: $\mathbf{x}' = \mathbf{V}\,\mathbf{x}\,\mathbf{V}^{-1}$. If translation and rotation are outermorphisms in 4D GA, then, of course, they are also outermorphisms in 5D GA. Hence, if we construct an outermorphism operator **M** from the versor **V**, we could instead use $\mathbf{x}' = \mathbf{Mx}$.

Line-plane intersection. To compute the intersection point **f** of a line **l** and a plane **p**, we use the general equation for intersecting subspaces: $\mathbf{f} = \mathbf{p}^* \bullet \mathbf{l}$. We can use this construction (the inner product of one primitive and the dual of the other) to compute the intersection of any pair of primitives. Even when the primitives don't intersect, the product will give a geometrically sensible answer that describes their incidence relationship.Because the line and the plane intersect each other at a point **q** *and* at infinity, **f** is a grade 2 object, a so-called flat point. This means that **f** is of the form $\mathbf{f} = \mathbf{q} \wedge \mathbf{e}_\infty$. Although it's often possible to continue computations directly with **f**, we could extract **q** from **f**. Formally, we can use the following for this purpose:

$$\mathbf{s}^* = \mathbf{e}_0 \bullet \mathbf{f} \qquad (10)$$
$$\mathbf{q} = \mathbf{s}^*\,\mathbf{e}_\infty\,\mathbf{s}^{*-1} \qquad (11)$$

$$\mathbf{q}_n = \frac{\mathbf{q}}{\mathbf{e}_\infty \bullet \mathbf{q}} \qquad (12)$$

In Equation 10, we first construct the dual of a sphere $\mathbf{s}^*$ with center point $\mathbf{q}$, through an arbitrary point ($\mathbf{e}_0$ in this case). In Equation 11, we reflect the point at infinity in the sphere to find its center point $\mathbf{q}$. In Equation 12 we normalize the point. In our ray-tracer implementation however, we more efficiently extract $\mathbf{q}$ from $\mathbf{f}$ by manipulating coordinates directly.

Line-sphere intersection. A line $\mathbf{l}$ and a sphere $\mathbf{s}$ intersect each other in a point pair or 1D circle. Computing this point pair $\mathbf{r}$ is similar to computing the intersection point of a line and a plane: $\mathbf{r} = \mathbf{s}^* \bullet \mathbf{l}$. We can check to see if the line and the sphere actually intersect by computing the radius squared ρ^2 of the 1D circle:

$$\rho^2 = \frac{\mathbf{r}^2}{(\mathbf{e}_\infty \wedge \mathbf{r})^2}$$

If ρ^2 is positive, the line and the sphere intersect. If ρ^2 is negative, the line and the sphere don't intersect. If necessary, we can recover the two individual intersection points $\mathbf{q}_-$ and $\mathbf{q}_+$ from $\mathbf{r} = \mathbf{q}_- \wedge \mathbf{q}_+$ using this equation:

$$\mathbf{q}_\pm = \frac{\pm\sqrt{\mathbf{r}\bullet\mathbf{r}} + \mathbf{r}}{\mathbf{e}_\infty \bullet \mathbf{r}}$$

Reflection. We reflect a line $\mathbf{l}$ in a plane $\mathbf{p}$ as follows: $\mathbf{l}' = \mathbf{plp}^{-1}$. This equation gives us a direct answer, even though $\mathbf{p}$ and $\mathbf{l}$ can be located anywhere in 3D space. We don't need to compute the intersection point of the line and the plane explicitly as needed in the other models.

Snell's law. Implementing Snell's law is straightforward in the model. Given a line $\mathbf{l}$, a plane $\mathbf{p}$, and refractive index η, we first compute a normal line $\mathbf{l}_n$. This line $\mathbf{l}_n$ is orthogonal to $\mathbf{p}$, and it runs through the intersection point of $\mathbf{l}$ and $\mathbf{p}$:

$$\mathbf{l}_n = \left(\frac{\mathbf{p}^* \bullet \mathbf{l}}{(\mathbf{e}_\infty \wedge \mathbf{e}_0) \bullet (\mathbf{p}^* \bullet \mathbf{l})} \bullet \mathbf{p} \right)^*$$

The refracted line is then computed:

$$\mathbf{l}' = \left(\operatorname{sign}(\mathbf{l}\bullet\mathbf{l}_n)\sqrt{1-\eta^2+(\mathbf{l}\bullet\mathbf{l}_n)^2\eta^2} - (\mathbf{l}\bullet\mathbf{l}_n)\eta \right)\mathbf{l}_n + \eta\mathbf{l}$$

Compare this equation to Equation 3, a similar formula that merely computes the directional aspect, while here we work directly with lines in space.

Implementation and performance

We implemented the ray tracers starting with the 5D GA model, because we were curious about its performance. We derived all other implementations from this. We didn't attempt to optimize any implementation to the extreme. Instead, we applied equal effort to each implementation to attempt to make a fair comparison of their performance.

Further Resources

As an accompaniment to information in this article, we've constructed a Web page at http://www.science.uva.nl/~fontijne/raytracer providing:

- nine implementations of the ray-tracing algorithm;
- a ray-tracing algorithm specification;
- more detailed benchmarks comparing the Geometric Algebra Implementation Generator (Gaigen) to CLU, another C++ package;
- two tables summarizing all equations used in this article;
- tutorials;
- Gaigen, including papers, documentation, and software; and
- links to other geometric-algebra-related software and resources.

Linear algebra implementation

We implemented the 3D and 4D LA classes in an object-oriented manner, taking efficiency into consideration. They don't use single instruction, multiple data instructions. We took parts of the 4D Plücker coordinates code directly from code generated by the Geometric Algebra Implementation Generator (Gaigen), our own C++ GA package.[9] But that code could just as easily have been copied from a textbook on projective geometry, such as Stolfi.[10]

Geometric algebra implementation

We implemented the models that use geometric algebra using Gaigen—an efficient, geometric algebra implementation that is publicly available (see the "Further Resources" sidebar). The Gaigen program can generate optimized C++ implementations of specific geometric algebras according to user specifications. It's our first attempt at implementing GA efficiently. GA seems so general that it's difficult to write a single efficient implementation (for example, a C++ template class) that implements every specific GA. However, we have seen one implementation called Boost::Clifford that uses a technique called *metaprogramming* (a smart use of C++ template classes) that is more efficient than Gaigen. Unfortunately, at the time of this writing it was not mature enough to use in the ray-tracer benchmarks. Gaigen can generate C++ source code for a specific GA for a specific application. Gaigen's user interface lets the user specify the properties of the GA required for the application and generate the source code to implement that specific algebra. The algebra properties include name, dimensionality, signature, required products, required functions, optimizations, and coordinate storage procedure.

Besides automatically generated code, another key idea behind Gaigen's efficiency is that it tracks the grade part usage for each multivector. Most objects that we use in GA occupy only certain grade parts (a vector is always grade 1, bivector is always grade 2). Because we know grade part usage, Gaigen doesn't have to store the coordinates of empty grade parts. This saves memory and computation time because no time is spent multiplying and adding values that are equal to zero anyway.

For even more efficiency, Gaigen lets users add optimizations for specific products of specific objects. Imag-

Table 1. Performance benchmarks.*

Model	Implementation	Full Rendering Time ($\times$ 23.3 seconds)	Rendering Time without BSP ($\times$ 0.99 seconds)	Executable Size (Kbytes)	Runtime Memory Use (Mbytes)
3D LA	Standard	1.00	1.00	52	6.2
3D GA	Gaigen	2.56	1.86	64	6.7
4D LA	Standard	1.05	1.22	56	6.4
4D GA	Gaigen	2.97	2.62	72	7.7
5D GA	Gaigen	5.71	4.58	100	9.9

*Tests were run on a Pentium III, 700-MHz notebook computer with a 256-Mbyte memory, running Windows 2000. We compiled programs using Visual C++ 6.0. We dynamically linked all support libraries, such as fltk, libpng, and libz, to get the executable size as small as possible. We measured runtime memory use with the task manager.

ine that the inner product of a 3-blade and a 2-blade is used 50 percent of the time in your application, users tell Gaigen to implement that product efficiently and regenerate the source code. To assist in this optimization process, Gaigen can profile the application at runtime and report which products it should optimize. Gaigen can read this report into its user interface to perform automatic optimizations.

Performance

Table 1 presents our benchmark results for each implementation of each model. Two columns contain rendering times; one with and one without time spent on line-BSP intersection computations. We show this separation because computation of the intersection point of lines and polygonal models (stored in BSP trees) dominate the full rendering time. We wrote a ray tracer because we wanted to benchmark a good mix of geometric computations. But it turned out that the application computes line-BSP tree intersections most of the time, which uses only a few types of geometric computations. Thus, we added an (optional) preprocessing phase to the ray tracer. For every pixel, this phase traces the spawned ray(s) through the scene and stores partial information about it in a data structure. The partial information only states what face of what model every ray intersects first. The actual rendering phase uses this information, and thus we can measure the rendering time in isolation from the time spent on BSP computations. Isolating the combinatorics of the intersection computations from the rest of the application provides two application benchmarks from one run. The full ray-tracing algorithm application spends its time mainly on line-BSP tree intersection tests. The other application performs a mix of geometric computations.

As the table shows, there's quite a difference between the two sets of benchmarks. Thus, you should interpret these benchmarks as an indication of the relative performance of the models. The precise performance figures will vary from implementation to implementation, platform to platform, and algorithm to algorithm.

Discussion

The 5D GA conformal model is the clear winner in the elegance contest. This model directly represents all geometric primitives using elements of the algebra. It reduces all geometric computations to elementary equations. The model lets us use circles and spheres as direct elements of computation, and we expect that this will have many advantages in other applications. The two less elementary Equations 10 and 12—used to extract points from bivectors—are 5D GA's only drawback. Further work might provide methods to avoid these computations, but this is still an open issue.

Performance-wise, the 5D GA model is the big loser; it's about five times slower than the most efficient models and about two times slower than the other GA methods. This is partly due to some areas in Gaigen that need improvement, and partly due to the model itself, which in some cases uses more computations or coordinates than other models. Still, we are representing 3D geometry in a 5D space, of which the geometric algebra requires a $2^5 = 32$ dimensional basis. Linear operations in that space would be 32×32 matrices that can also be performed in the 3D LA model using 3×3 matrices. Compared to the expected loss of efficiency of $32{\times}32/3{\times}3 = 110\times$, achieving a five times slow down is not a bad result. We're currently investigating methods to improve the Gaigen's performance in general and the model specifically, and we might implement these in Gaigen's next version. These methods include data structure improvement, coordinate usage tracking at the subgrade level, and automatic simplification of expressions at the symbolical and coordinate levels. Another possible approach is to use single instruction, multiple data instruction sets better fitted for the model, but it will probably be a long time before general-purpose CPUs can efficiently handle geometric algebra.

By contrast, let us consider the most basic models, 3D LA and 3D GA. Judging by the equations alone, in this particular application, 3D GA offers no great advantages over 3D LA. Although, 3D GA's reflection Equation 4 is nicer than 3D LA's Equation 2, since it's shorter, requires only one type of product, and also works in other cases (for example, reflecting a bivector). With 3D GA we can use and construct rotors (that is, quaternions) more naturally and derive some equations more easily, but these are its only advantages over 3D LA and that does seem not enough to justify its use. However, some definite advantages will become obvious once practitioners become more familiar with GA. As discussed in Goldman,[1] the 3D LA and 4D LA models use the same vectors to represent many objects (directions, points, normal vectors, and so on). The subtle differences in the way these vectors add and transform can lead to mysterious problems and difficult to trace bugs. Switching to GA automatically resolves many of these problems. The grade mechanism of GA can represent higher dimensional subspaces as direct elements of computation. This lets us distinguish between objects that would otherwise appear the same, but act differently. A subjective advantage of GA that we can't uncover by considering the equations alone, is the better understanding of geometry that might be gained by learning GA. We benefited from this even while implementing the 3D LA and 4D LA models—for example in the derivation of Equation 3 and the use of Plücker coordinates.

When we look at the 3D models' performances, the 3D GA model using Gaigen is about two times slower than 3D LA. There is no fundamental reason why this should be so; virtually the same computations are made in both GA and LA in the 3D models. The main cause for the lower performance of GA is Gaigen's soft typing of the geometric algebra objects at compile time; Gaigen represents all types of objects (scalars, vectors, bivectors, trivectors, rotors and so on) by a single data type. Before computing a product or operation, Gaigen checks the grade usage of the argument(s) and then acts accordingly. This conditional step between function call and actual computation is largely responsible for the drop in performance. Experimental benchmarks suggest a raw performance increase between five and ten times is possible when GA objects are strongly typed at compile time.

The increase of elegance due to using GA instead of LA in the 4D model is most obvious in primitive construction, outermorphism use for rotation/translation, and general intersection equation use. Some of these improvements (like the outermorphism) could be used (and probably are used by some) in the 4D LA model. Because of the difficulty in understanding the 4D LA model, there is no widespread use of such techniques. Understanding GA is necessary before practitioners can add these techniques to the 4D LA model. This would, in essence, incorporate more of GA into the traditional model, which already contains elements that do not strictly belong there—for example, Plücker coordinates and quaternions. Unfortunately, due to deficiencies in both 4D models, other geometric computations, like reflection, are even more awkward to implement than in the 3D models. The 5D GA conformal model resolves most of these problems. Gaigen causes a performance drop in the 4D models by a factor of about 3, but as previously discussed, future GA implementations should reduce this to a small performance penalty, presumably less than one and a half times.

Conclusion

This comparison demonstrates that there is a sliding scale between these models, with performance on one end and elegance on the other. For now, the traditional models (3D LA and 4D LA) remain most efficient and are most appropriate in time-critical applications.

For all other applications, such as experiments, prototypes, one-time offline tools, and so on, we would have liked to recommend using the elegant 5D GA conformal model to tackle geometric problems. However, this model isn't fully mature yet, although we expect this growth to occur in the next two years. Currently there are no books and few practical papers that describe this model. But we, as well as others, are exploring it theoretically, practically, and educationally to make it usable for the computer science community. We recommend study of the model now and keeping informed of research developments to prepare to possibly reap its benefits in the near future.

In between these extremes of elegance and performance, the 3D and 4D GA models are useful for study, experimentation, improved insight into geometry, and implementation of more advanced geometric problems. Just because we observed no great improvement in the 3D model's elegance in this particular ray-tracing application, doesn't mean that other applications won't benefit from GA. The 4D GA model is especially useful in practice. It offers a more natural path towards understanding Plücker coordinates and projective geometry, and it is a good source of new techniques and even code. In our implementation of the 4D LA model, we directly copied code (fault free and automatically generated by Gaigen) from the 4D GA implementation to the 4D LA implementation. For application programmers, this method might a place for GA in their suite of techniques, that is, to generate better LA code. But we expect that many will eventually program directly in GA. ■

Acknowledgement

Our sincere thanks to Stephen Mann for many useful comments and suggestions.

References

1. R. Goldman, "Illicit Expressions in Vector Algebra," *ACM Trans. Graphics*, vol. 4, no. 3, July 1985, pp. 223-243.
2. R. Goldman, "On the Algebraic and Geometric Foundations of Computer Graphics," *ACM Trans. Graphics*, vol. 21, no. 1, Jan. 2002, pp. 52-86.
3. S. Mann, N. Litke, and T. DeRose, *A Coordinate Free Geometry ADT*, research report CS-97-15, Computer Science Dept., Univ. of Waterloo, 1997.
4. T. DeRose, *Coordinate-Free Geometric Programming*, tech. report 89-09-16, Dept. of Computer Science, Univ. of Washington, Sept. 1989.
5. L. Dorst and S. Mann, "Geometric Algebra: A Computation Framework for Geometrical Application, Part 1," *IEEE Computer Graphics and Applications*, vol. 22, no. 3, May/June 2002, pp. 24-31.
6. S. Mann and L. Dorst, "Geometric Algebra: A Computation Framework for Geometrical Application, Part 2," *IEEE Computer Graphics and Applications*, vol. 22, no. 4, July/Aug. 2002, pp. 58-67.
7. D. Hestenes, H. Li, and A. Rockwood, "A Unified Algebra-

ic Framework for Classical Geometry," *Geometric Computing with Clifford Algebra*, G. Sommer, ed., Springer, 1999, http://modelingnts.la.asu.edu/html/UAFCG.html.

8. A.S. Glassner, ed., *An Introduction To Ray Tracing*, Academic Press, 1989.
9. D. Fontijne, T. Bouma, and L. Dorst, *Gaigen: a Geometric Algebra Implementation Generator*, http://www.science.uva.nl/~fontijne/raytracer.
10. J. Stolfi, *Oriented Projective Geometry*, Academic Press, 1991.

Daniel Fontijne *is a scientific programmer at the University of Amsterdam. His research interests include creation of an efficient implementation of geometric algebra for use in computer graphics, computer vision, and robotics. He has an MSc in artificial intelligence from the University of Amsterdam.*

Leo Dorst *is an assistant professor at the Informatics Institute at the University of Amsterdam. His research interests include geometric algebra and its applications to computer science. He has an MSc and PhD in applied physics of computer vision from Delft University of Technology.*

Readers may contact Daniel Fontijne and Leo Dorst at the Informatics Inst., Univ. of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, Netherlands, email {fontijne, leo}@science.uva.nl.