Performance and elegance of five models of 3D Euclidean geometry in a ray tracing application [*]

Daniel Fontijne and Leo Dorst
University of Amsterdam

February 5, 2003

1 Abstract

Computations of 3D Euclidean geometry can be performed using various computational models of different effectiveness. In this paper we compare five alternatives: 3D linear algebra, 3D geometric algebra, a mix of 4D homogeneous coordinates and Plücker coordinates, a 4D homogeneous model using geometric algebra, and the 5D conformal model using geometric algebra. Higher dimensional models and models using geometric algebra can express geometric primitives, computations and constructions more elegantly, but this elegance may come at a performance penalty.

We explore these issues using the implementation of a simple ray tracer as our practical goal and guide. We show how to implement the most important geometric computations of the ray tracing algorithm using each of the five models and benchmark each implementation. [1]

2 Introduction

The space we live in is quite well described as a 3-dimensional Euclidean geometry for most computer graphics applications. Although it would seem straightforward to directly implement this for the generation of realistic images and simulation of objects and their properties, most of us find a more indirect method more attractive: we construct a computational model of the 3D geometry, and implement that. This often improves our programs in structure and efficiency. An example is the wide-spread use of homogeneous coordinates, which uses a 4D linear algebra to do some of 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 6-dimensional space).

[*] Suggestion for short CG&A title: Modeling 3D Euclidean geometry.

[1] Suggestion for 25 word CG&A abstract: 3D Euclidean geometry can be modeled in several ways. We compare the elegance and performance of five such models in a ray tracing application.

In fact, the choice of models is getting confusing. Explanatory papers have been written [6] [7] [8] [9], often suggesting different algebras for different aspects of geometry. Our programs typically reflect that. The recently discovered *geometric algebra* [1a] [1b] [5] appears to be just one more possibility (groan), but that is not the right way of looking at it. Instead, in geometric algebra we finally have a framework containing all the options, neatly laid out. It cleans up the situation by assigning various 'tricks' such as quaternions and Plücker coordinates to a proper geometric algebra of appropriate real, interpretable vector spaces.

This paper gives a comparison of five models of 3D Euclidean geometry – not theoretically, but by showing how you would implement a simple recursive ray tracer in each of them. It is 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 paper can be considered to be a practical sequel to two tutorials on geometric algebra previously published in CG&A [1a] [1b], and we will make frequent references to those tutorials.

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*, i.e. the familiar homogeneous coordinates); the 4D homogeneous model (*4D GA*, a geometric algebra which naturally absorbs Plücker coordinates of lines and planes into homogeneous computations); and the 5D conformal model (*5D GA*, a geometric algebra which is new – it 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 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 is used to demonstrate what kind of improvements are possible with more sophisticated models. Although in this paper we don't make explicit use of Grassmann spaces as recommended by [7], but we shall see that using geometric algebra to implement Grassmann spaces significantly extends their applicability.

Our reasons for choosing a ray tracer as benchmark are the following. 1) Everybody familiar with computer graphics knows how a basic ray tracer works, and possibly has implemented one. 2) Implementing the core of a ray tracer can be done with relatively little code, which was important to us, since we were going to write many different implementations of the same algorithm. 3) A ray tracer contains a diverse selection of geometric computations, like rotation, translation, reflection, refraction, (signed) distance computation, and line-plane and line-sphere intersection computations. This allows us to show by example how to perform these computations in different models. But we emphasize that our main goal in this paper 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 is sufficiently sophisticated to render images such as figure 1.

In the following, we first discuss the basic algorithm of the ray tracer and the geometric computations required to implement it. After that, we briefly introduce each of the five models and show in detail how to implement five of the most important geometric computations required by the ray tracer. Then

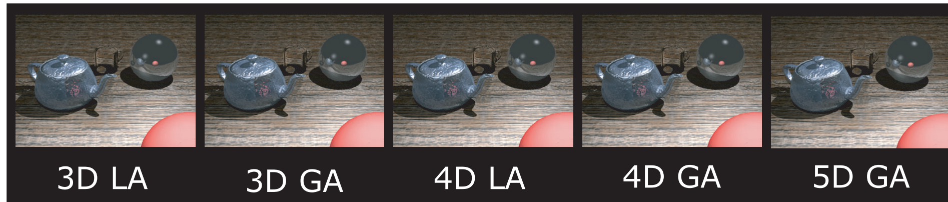

Figure 1: The same result can be achieved in many ways. These images are identical, but each one was rendered using a different model of 3D Euclidean geometry. The scene consist of 5 objects modeled with about 7800 triangles: a textured/bumpmapped teapot, a transparent drinking glass, a reflective sphere, a red diffuse sphere, and a textured/bumpmapped piece of wood.

it is time for a short section on the implementation of the algebras, followed by the performance results of each ray tracer implementation. We end with a discussion of the relative performance and elegance of the models.

3 The ray tracer

We use a basic recursive ray tracing algorithm, without special techniques for efficiency, except for the use of a BSP tree to accelerate ray-mesh intersection computations. Describing the precise algorithm in great detail is not meaningful here, since only the geometric computations matter to the discussion at hand. A more detailed specification is available elsewhere[3].

The ray tracing algorithm accepts as input a description of the scene, including camera, lighting and polygonal model information, like 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, lighting computations are performed for each visible or ambient light source. The final color of each pixel is determined by the weighted sum of such lighting computations.

The ray tracing algorithm requires representations of geometric primitives like vectors, points, lines, planes, spheres, as well as transformations of these primitives. In section 4 we show how to represent these primitives and operators in each model of geometry. In some models, primitives can also be operators. For instance using geometric algebra, a plane can be 'applied' to another primitive directly in order to reflect that primitive in the plane.

The geometric computations and operations that we must be able to implement in each model in order 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, of lines and triangles, and of lines and spheres,
- computation of the angle between lines and/or the angle between planes,

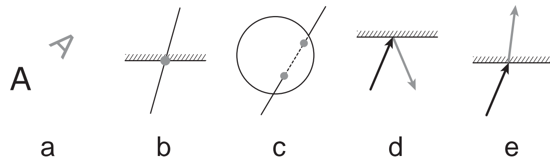

Figure 2: Illustration of the geometric computations that will be treated in detail for each model. Input of the computations is shown in black, output in grey. a: translation and rotation. b: intersection of a line and a plane. c: intersection of a line and a sphere. d: reflection of a directed line in a plane. e: refraction of a directed line in a plane.

- computation of the distance between two points and the signed distance of points to planes.

In the next section, a selection of these operations is treated in detail for each model.

At this point we would have liked to give a more detailed specification of each of the geometric computations we are about to discuss. The problem is that to write down those very descriptions already implies the use of a specific model, since using a model is the only method we know to precisely encode geometry. An important theme of this paper is that the use of any model, even a model of 3D Euclidean geometry, not only determines how you implement your solution, but also shapes the way you think about the problem. So if we want to remain impartial with respect to the five models, we should not use one of those models at this point to specify the geometric computations. Instead we have chosen to include a graphical representation in figure 2. The icons in that figure show only the relevant geometric primitives. As the reader will see shortly, derived geometric primitives like angles, intermediate intersection points and surface normals arise from the way we are used to implement the computations in mainstream models of 3D Euclidean geometry, and do not necessarily arise in other models. For example, when we treat the conformal model, a directed line can be reflected in a plane without ever using a surface normal or the intersection point of the line and the plane.

4 The models

In the following five sections we give a short informal introduction of each of the five models. These introductions show only how some important primitives and rotation/translation operations are represented. For the novel GA models, we give references to sources where you can learn more about them. After each introduction, we show the equations used to implement the five geometric computations from figure 2. Readers with little mathematical background should not be discouraged by these equations that make up the bulk of the next section. Instead we encourage all readers *not* to focus on understanding the equations, but to read with a bird's eye view: please skip back and forth between the five sections and compare the length and simplicity of the equations and the number of split-up cases.

Knowing that for each geometric operation in each model there are alternative ways to implement them, we have always tried to 'do the sensible thing' in each model. The equations we use to implement the geometric operations in the 3D LA model are virtually identical to those quoted in [4] as most efficient.

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

4.1 3D linear algebra

In the 3D LA model, all primitives are represented using vectors and scalars. A point is represented by a vector that points from the origin to the location of the point. A line is represented by a vector that points from the origin to some point on the line, and a unit vector that points along the direction of the line. A plane is represented by a normal vector and a scalar that gives the distance of the plane to the origin. A sphere is represented by a vector that points from the origin to the center of the sphere and a scalar that gives the radius of the sphere. Note that every primitive is explicitly represented relative to a specific origin that is chosen a priori.

Translation is represented by a vector. Rotation about the origin is a linear mapping, so this can be represented by a 3×3 matrix.

All geometric computations are made using matrix-vector multiplication, addition and subtraction of vectors, scalar multiplication, dot products (denoted by the $\cdot$ symbol) and cross products (denoted by the $\times$ symbol).

4.1.1 Rotation/translation

A point $\mathbf{q}$ is rotated/translated as follows:

$$\mathbf{q}' = \mathbf{R}\mathbf{q} + \mathbf{t} \tag{1}$$

where $\mathbf{R}$ is a rotation matrix, and $\mathbf{t}$ is a translation vector. To translate/rotate a line (given by a point $\mathbf{q}_l$ on the line and a unit vector $\mathbf{u}$ along the line) one computes

$$\mathbf{q}_l' = \mathbf{R}\mathbf{q}_l + \mathbf{t} \tag{2}$$

$$\mathbf{u}' = \mathbf{R}\mathbf{u} \tag{3}$$

A plane (given by its unit normal vector $\mathbf{n}$ and the scalar distance to the origin δ) is rotated/translated by

$$\mathbf{n}' = \mathbf{R}\mathbf{n} \tag{4}$$

$$\delta' = \delta + \mathbf{t}\cdot\mathbf{n}' \tag{5}$$

4.1.2 Line-plane intersection

The intersection point $\mathbf{q}_i$ of a line (given by a point $\mathbf{q}_l$ on the line and a unit vector $\mathbf{u}$ along the line) and a plane (given by its unit normal vector $\mathbf{n}$ and the scalar distance to the origin δ) can be computed as:

$$\mathbf{q}_i = \mathbf{q}_l - \frac{((\mathbf{q}_l \cdot \mathbf{n}) - \delta)\,\mathbf{u}}{\mathbf{u} \cdot \mathbf{n}} \tag{6}$$

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

4.1.3 Line-sphere intersection

The two intersection points $\mathbf{q}_-$ and $\mathbf{q}_+$ of a line (given by a point $\mathbf{q}_l$ on the line and a unit vector $\mathbf{u}$ along the line) and a sphere (given by its center point $\mathbf{q}_s$ and its scalar radius ρ) can be computed using the following equations. First the closest point $\mathbf{q}_c$ to the center of the sphere on the line is computed as:

$$\mathbf{q}_c = \mathbf{q}_l + ((\mathbf{q}_s - \mathbf{q}_l) \cdot \mathbf{u})\,\mathbf{u} \tag{7}$$

then the normalized squared Euclidean distance δ_n^2 of $\mathbf{q}_c$ to $\mathbf{q}_s$ can be used to determine if the line intersects the sphere:

$$\delta_n^2 = \frac{(\mathbf{q}_c - \mathbf{q}_s) \cdot (\mathbf{q}_c - \mathbf{q}_s)}{\rho^2} \tag{8}$$

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

$$\mathbf{q}_\pm = \mathbf{q}_c \pm \rho\sqrt{1 - \delta_n^2}\,\mathbf{u} \tag{9}$$

4.1.4 Reflection

The reflected direction $\mathbf{u}'$ of a line (given by a point $\mathbf{q}_l$ on the line and a unit vector $\mathbf{u}$ along the line) in a plane (given by its normal vector $\mathbf{n}$), can be computed as:

$$\mathbf{u}' = -2(\mathbf{n} \cdot \mathbf{u})\mathbf{n} + \mathbf{u} \tag{10}$$

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 6) before we obtain a full representation of the reflected line.

4.1.5 Snell's law

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

$$\frac{\sin\phi_1}{\sin\phi_2} = \frac{\eta_2}{\eta_1} \tag{11}$$

where ϕ_1 is the incoming angle, ϕ_2 is the outcoming angle, and η_1 and η_2 are the refractive indices of the media. In appendix A we use geometric algebra to compactly derive the classical equation for implementing Snell's law. Here we just 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 $\eta = \frac{\eta_2}{\eta_1}$, the refractive index of medium 2 relative to medium 1. This is all we need to compute the refracted direction of the line:

$$\mathbf{u}' = \left(\text{sign}\left(\mathbf{n} \cdot \mathbf{u}\right) \sqrt{1 - \eta^2 + \left(\mathbf{n} \cdot \mathbf{u}\right)^2 \eta^2} - \left(\mathbf{n} \cdot \mathbf{u}\right) \eta\right) \mathbf{n} + \eta \mathbf{u} \tag{12}$$

4.2 3D geometric algebra

3D geometric algebra is an extension of 3D linear algebra [1a] [1b]. It has an operation to span subspaces through the origin: the *outer product* ([1a], pg. 25) denoted by the $\wedge$ symbol. Such subspaces or *blades* ([1a], pg. 25) are then 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 much better under linear transformations. We can naturally extend the inner product (denoted by the $\cdot$ symbol) to blades, and this is useful for projection and metric relationships.

GA also has a *geometric product* ([1a], pg. 27), denoted by a half space symbol, as in $\mathbf{a}\,\mathbf{b}$. The geometric product permits multiplication and *division* ([1a], pg. 30) by vectors and subspaces. The ratio of two vectors forms a *rotor* ([1b], pp. 58-60), which can be used as a rotation operator. In fact, it has the same properties as a quaternion, but within the context of geometric algebra it is a real operator that can moreover be used to rotate subspaces of any grade. Alternatively, a rotor can be constructed 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 ([1b], pp. 60-61), denoted by a superscript *, returns the dual of any blade, i.e. the orthogonal complement in 3D space. All of these construction naturally extend to n-dimensional vector spaces.

A 'general number' or *multivector* in 3D GA can be represented by 8 coordinates relative to an 8-dimensional basis: 1 coordinate for scalars, 3 coordinates for vectors, 3 coordinates for bivectors and 1 coordinate for trivectors (3-blades, interpreted as volume elements).

4.2.1 Rotation/translation

Rotation of a vector about an axis through the origin is done like this in 3D GA:

$$\mathbf{v}' = \mathbf{R}\,\mathbf{v}\,\mathbf{R}^{-1} \tag{13}$$

The vector is *sandwiched* between the rotor $\mathbf{R}$ and its inverse $\mathbf{R}^{-1}$. $\mathbf{R}$ is created as $\mathbf{R} = \exp\left(-\frac{1}{2}\phi\mathbf{b}\right) = \cos\frac{1}{2}\phi - \mathbf{b}\sin\frac{1}{2}\phi$, where ϕ is the angle of rotation and $\mathbf{b}$ the unit bivector denoting the plane of rotation. Such an $\mathbf{R}$ is normalized. This implies that $\mathbf{R}^{-1}$ is equal to $\mathbf{R}^{\sim}$,the reverse of $\mathbf{R}$ ([1a], pg. 30). The reverse can be computed efficiently by sign flipping part of the coordinates of $\mathbf{R}$.

Sandwiching operations like $\mathbf{R}\,\mathbf{v}\,\mathbf{R}^{-1}$ are common in GA. They are typically used to apply objects like rotors to blades. Once you replace rotation matrix multiplication by this rotor sandwiching operation, points (represented

by a vector $\mathbf{q}$ from the origin to the point) and lines (represented by a point $\mathbf{q}_l$ and a direction $\mathbf{u}$) are transformed the same way in 3D GA as they are in 3D LA

A plane is now given by its bivector $\mathbf{b}$ and its scalar distance to the origin δ. It is rotated/translated as follows:

$$\mathbf{b}' = \mathbf{R}\,\mathbf{b}\,\mathbf{R}^{-1} \tag{14}$$

$$\delta' = \delta + (\mathbf{t} \wedge \mathbf{b}')^* \tag{15}$$

$(\mathbf{t} \wedge \mathbf{b}')^*$ is equal to $\mathbf{t} \cdot (\mathbf{b}'^*)$, but slightly more efficient.

4.2.2 Line-plane intersection

The intersection point $\mathbf{q}_i$ of a line (given by a point $\mathbf{q}_l$ on the line and a unit vector $\mathbf{u}$ along the line) and a plane (given by a unit bivector $\mathbf{b}$ and a scalar distance to the origin δ) can be computed as:

$$\mathbf{q}_i = \mathbf{q}_l - \frac{\big((\mathbf{q}_l \wedge \mathbf{b})^* - \delta\big)\,\mathbf{u}}{(\mathbf{u} \wedge \mathbf{b})^*} \tag{16}$$

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

4.2.3 Line-sphere intersection

Line-sphere intersection is handled entirely the same way in 3D GA as in 3D LA, except that the dot products are replaced by equivalent inner products.

4.2.4 Reflection

The reflected direction $\mathbf{u}'$ of a line (given by a point $\mathbf{q}_l$ on the line and a unit vector $\mathbf{u}$ along the line) in a plane (given by its unit bivector $\mathbf{b}$ and its distance to the origin δ), can be computed as:

$$\mathbf{u}' = -\mathbf{b}\,\mathbf{u}\,\mathbf{b}^{-1} = \mathbf{b}\,\mathbf{u}\,\mathbf{b} \tag{17}$$

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

4.2.5 Snell's law

Snell's law is implemented the same way in the 3D GA model as in the 3D LA model. To implement Snell's law in the 3D GA model, we only have to set $\mathbf{n} = \mathbf{b}^*$ (where $\mathbf{b}$ is the unit bivector of the plane), and implement the rest as in 3D LA equation 12.

4.3 4D homogeneous coordinates, Plücker coordinates, 4D linear algebra

This model is the most incoherent of all models presented in this paper, though it is probably representative for what an advanced computer graphics programmer would use today. The model uses homogeneous coordinates, Plücker coordinates and 4×4 transformation matrices to implement part of an oriented projective geometry, such as described in [10].

In homogeneous coordinates an extra basis vector or axis is used, besides the standard x, y and z axis. This axis is usually called w and is used to get rid of the origin as a special point relative to which other primitives are described. It allows one to represent arbitrary affine subspaces (i.e. lines and planes floating in space) as elements of direct computation.

The 4D homogeneous coordinates of a point $\mathbf{q}$ 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. A point can thus be denoted $\mathbf{q} = (\vec{q} : 1)$. 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 of [7].

Plücker coordinates are the homogeneous coordinates of lines and planes and they can be useful for intersection and relative orientation computations. In the next section, we show that 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 used too little and are underappreciated.

The Plücker coordinates of a line $\mathbf{l}$ through two points $\mathbf{q}_1 = (\vec{q}_1 : 1)$ and $\mathbf{q}_2 = (\vec{q}_2 : 1)$ are

$$\mathbf{l} = (\vec{q}_1 - \vec{q}_2 : \vec{q}_1 \times \vec{q}_2) = (\vec{q}_1 - \vec{q}_2 : (\vec{q}_1 - \vec{q}_2) \times \vec{q}_1) \tag{18}$$

So six coordinates, that can be grouped into two 3-vectors, represent a line as illustrated in figure 3.

The Plücker coordinates of a plane are the normal vector $\vec{n}$ of the plane and its scalar distance to the origin δ:

$$\mathbf{p} = [\vec{n} : \delta] \tag{19}$$

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 product and special 'Plücker products'. To do the Plücker products we often have to take the coordinates apart into scalars and 3D vectors, perform some computations on them, and then reassemble them into a Plücker object again. 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 $(\vec{v}\,' : 0) = \mathbf{M}(\vec{v} : 0)$.

4.3.1 Rotation/translation

A point $\mathbf{q}$ is rotated/translated through multiplying it by a 4×4 transformation matrix $\mathbf{M}$:

$$\mathbf{q}' = \mathbf{M}\mathbf{q} \tag{20}$$

Such a simple product between the affine transformation matrix $\mathbf{M}$ and the Plücker coordinates of a line $\mathbf{l}$ 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 those, and reconstruct the line:

$$\mathbf{l} = (\vec{u} : \vec{v}) \tag{21}$$

$$\mathbf{q}' = (\vec{q}\,' : 1) = \mathbf{M}(\vec{v} \times \vec{u} : 1) \tag{22}$$

$$\mathbf{l}' = (\mathbf{M}\vec{u} : \mathbf{M}\vec{u} \times \vec{q}\,') \tag{23}$$

A plane $\mathbf{p}$ can not be multiplied directly with a transformation matrix $\mathbf{M}$, but it can be derived that if $\mathbf{M}$ contains only rotations and translation, then

$$\mathbf{p}' = \mathbf{M}^{-T}\mathbf{p} \tag{24}$$

produces the transformed plane ($\mathbf{M}^{-T}$ is the inverse of the transpose of $\mathbf{M}$).

4.3.2 Line-plane intersection

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

$$\mathbf{q} = (\frac{\vec{v} \times \vec{n} + \delta\vec{u}}{\vec{u} \cdot \vec{n}} : 1) \tag{25}$$

In practice, equations like this one are implemented 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, such as in [10].

4.3.3 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 done differently: first $\mathbf{l}$ is translated over the vector $\vec{t} = -\vec{q}_s$ such that the center of the sphere is at the origin. Then the point on $\mathbf{l}$ closest to the center of the sphere closest can be computed:

$$\mathbf{l}_t = (\vec{u}_t : \vec{v}_t) = (\vec{u} : \vec{v} + \vec{u} \times \vec{t}) \tag{26}$$

$$\mathbf{q}_c = \left(\frac{\vec{v}_t \times \vec{u}_t}{|\vec{u}_t|} : 1\right) \tag{27}$$

The rest of the computations are identical to those in the 3D LA model:

$$\delta_n^2 = \frac{(\mathbf{q}_c - \mathbf{q}_s)\cdot(\mathbf{q}_c - \mathbf{q}_s)}{\rho^2} \tag{28}$$

$$\mathbf{q}_\pm = \mathbf{q}_c \pm \rho\sqrt{1 - \delta_n^2}\,\vec{u}_t \tag{29}$$

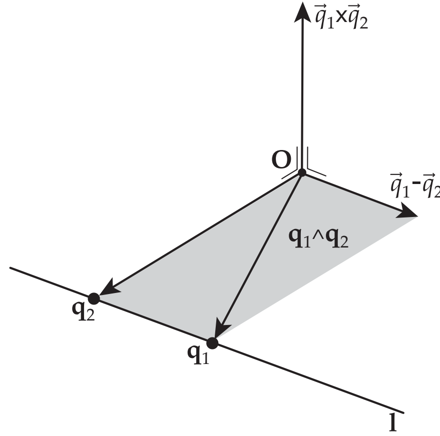

Figure 3: The Plücker coordinates of a line in 4D LA and 4D GA. A directed line $\mathbf{l}$ through two homogeneous points $\mathbf{q}_1 = (\vec{q}_1 : 1)$ and $\mathbf{q}_2 = (\vec{q}_2 : 1)$ can be fully described by its Plücker coordinates $(\vec{q}_1 - \vec{q}_2 : \vec{q}_1 \times \vec{q}_2)$. The vector $\vec{q}_1 - \vec{q}_2$ gives the direction of the line. The vector $\vec{q}_1 \times \vec{q}_2$ encodes both the distance of the line to the origin and the normal to the plane through the origin in which both $\mathbf{q}_1$ and $\mathbf{q}_2$ lie. In 4D GA, the single bivector $\mathbf{q}_1 \wedge \mathbf{q}_2$ (illustrated by the grey area) describes the entire line.

4.3.4 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

$$\vec{u}' = -2(\vec{n} \cdot \vec{u})\vec{n} + \vec{u} \tag{30}$$

and then construct a new line from the intersection point $\mathbf{q}$ of $\mathbf{l}$ and $\mathbf{p}$, and the reflected direction $\vec{u}'$. We have to explicitly compute $\mathbf{q}$ (using equation 25) before we obtain a full representation of the reflected line.

4.3.5 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.

4.4 4D homogeneous model using geometric algebra

In this section, we redo the previous section, this time using GA instead of LA. 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. ([1b] pp. 63-65) gives more details on the homogeneous model from a geometric algebra point of view.

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 $\mathbf{w}$. As with any Euclidean

unit vector, $\mathbf{e}_0 \cdot \mathbf{e}_0 = 1$. The $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ axis are called $\mathbf{e}_1$, $\mathbf{e}_2$ and $\mathbf{e}_3$. Points are defined as

$$\mathbf{q} = \vec{q} + \mathbf{e}_0 \tag{31}$$

where $\vec{q}$ is a Euclidean 3D vector that points from the origin to $\mathbf{q}$. 3D vectors by themselves are therefore represented by vectors with an $\mathbf{e}_0$ component of zero in the homogeneous model. 3D vectors can be added to points to produce translated points.

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

$$\mathbf{l} = \mathbf{q}_1 \wedge \mathbf{q}_2 \tag{32}$$

If we choose the appropriate bivector basis for our 4D GA, the 6 coordinates of $\mathbf{l}$ are exactly the Plücker coordinates of the line. See figure 3 for an illustration.

A plane $\mathbf{p}$ is constructed by wedging three points together (i.e. $\mathbf{q}_1$, $\mathbf{q}_2$ and $\mathbf{q}_3$):

$$\mathbf{p} = \mathbf{q}_1 \wedge \mathbf{q}_2 \wedge \mathbf{q}_3 \tag{33}$$

Again, with the appropriate trivector basis, the 4 coordinates of trivector $\mathbf{p}$ are identical to the Plücker coordinates of the plane.

We often use that $\mathbf{e}_0 \cdot \mathbf{l}$ and $\mathbf{e}_0 \cdot \mathbf{p}$ retrieve the direction elements of a line or plane, resulting in a purely Euclidean vector or bivector.

A linear transformation f (such as rotation, translation and projection) on vectors can be naturally made to act on blades (i.e. lines and planes) by demanding

$$f(\mathbf{a} \wedge \mathbf{b}) = f(\mathbf{a}) \wedge f(\mathbf{b}) \tag{34}$$

for all vectors $\mathbf{a}$ and $\mathbf{b}$. It is then called an *outermorphism*. If a transformation is an outermorphism, we can construct an outermorphism operator for it. The outermorphism operator is the matrix representation of the linear transformation. It can be used to transform any primitive (vector, point, line or plane). A 4×4 matrix is used to transform points, a 6×6 matrix for lines, and another 4×4 matrix for planes. The 4×4 matrix used to transform points and vectors is exactly the traditional 4×4 transformation matrix that is used in homogeneous coordinates. The construction of the outermorphism operator can be done naturally in GA, applying definition 34.

To do our geometric computations in this section, we use the standard set of GA products and operators as with 3D GA, plus the outermorphism operator.

4.4.1 Rotation/translation

Any primitive $\mathbf{x}$ can be rotated/translated applying outermorphism operator $\mathbf{M}$:

$$\mathbf{x}' = \mathbf{M}\mathbf{x} \tag{35}$$

There is no need to split this operation into different cases (point, vector, line or plane) as in the 4D LA model. Outermorphisms automatically handle each case correctly.

4.4.2 Line-plane intersection

The intersection point $\mathbf{q}$ of a line $\mathbf{l}$ and a plane $\mathbf{p}$ is given by

$$\mathbf{q} = \mathbf{p}^* \cdot \mathbf{l} \tag{36}$$

This is the standard primitive intersection equation in the homogeneous model. For example, it can also be used to compute the intersection of two planes, or even of two lines, given that dual (*) is computed with respect to the correct (sub-) space as detailed in ([1b], pg. 65). In general the point $\mathbf{q}$ will not be normalized; this can be enforced by dividing $\mathbf{q}$ by $\mathbf{e}_0 \cdot \mathbf{q}$.

4.4.3 Line-sphere intersection

As in the 4D LA model, we proceed by first computing the closest point $\mathbf{q}_c$ on the line $\mathbf{l}$ to the sphere (given by its center point $\mathbf{q}_s$ and its radius ρ). We translate $\mathbf{l}$ 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.

$$\mathbf{l}_t = \mathbf{l} - \mathbf{t} \wedge (\mathbf{e}_0 \cdot \mathbf{l}) \tag{37}$$

Note that $\mathbf{t}$ is a Euclidean vector and that in the 4D LA model we used the notation $\vec{t}$ for it. Here we use the $\mathbf{t}$ notation, since it is still a member of the algebra because 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:

$$\mathbf{q}_c = (\mathbf{e}_0 \cdot \mathbf{l}_t)\mathbf{l}_t^{-1} + \mathbf{t} \tag{38}$$

We can then proceed to compute the intersection points of the line and the sphere $\mathbf{q}^+$ and $\mathbf{q}^-$ as explained in the section on 3D LA:

$$\delta_n^2 = \frac{(\mathbf{q}_c - \mathbf{q}_s) \cdot (\mathbf{q}_c - \mathbf{q}_s)}{\rho^2} \tag{39}$$

$$\mathbf{q}_\pm = \mathbf{q}_c \pm \rho\sqrt{1 - \delta_n^2}\,(\mathbf{e}_0 \cdot \mathbf{l}) \tag{40}$$

4.4.4 Reflection

Unfortunately, the homogeneous model does not allow us to reflect an arbitrary line $\mathbf{l}$ in an arbitrary plane $\mathbf{p}$ 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 $\mathbf{l}$ and $\mathbf{p}$ over a vector $-\mathbf{t}$ such that their intersection point $\mathbf{q}$ 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 to where it should be:

$$\mathbf{q} = \frac{\mathbf{p}^* \cdot \mathbf{l}}{\mathbf{e}_0 \cdot (\mathbf{p}^* \cdot \mathbf{l})} \tag{41}$$

$$\mathbf{t} = \mathbf{q} - \mathbf{e}_0 \tag{42}$$

$$\mathbf{l}_t = \mathbf{l} - \mathbf{t} \wedge (\mathbf{e}_0 \cdot \mathbf{l}) \tag{43}$$

$$\mathbf{p}_t = \mathbf{p} - \mathbf{t} \wedge (\mathbf{e}_0 \cdot \mathbf{p}) \tag{44}$$

$$\mathbf{l}'_t = \mathbf{p}_t\,\mathbf{l}_t\,\mathbf{p}_t \tag{45}$$

$$\mathbf{l}' = \mathbf{l}'_t + \mathbf{t} \wedge (\mathbf{e}_0 \cdot \mathbf{l}'_t) \tag{46}$$

Note the simple equation we use to translate $\mathbf{l}$, $\mathbf{p}$ and $\mathbf{l}'_t$ in equations 37, 43, 44 and 46. It works for points, lines and planes.

4.4.5 Snell's law

The use of geometric algebra in the homogeneous model does not allows us to handle Snell's law more elegantly. So we still have to separate the incoming line $\mathbf{l}$ into its direction ($\mathbf{u} = \mathbf{e}_0 \cdot \mathbf{l}$) and its intersection point with the plane $\mathbf{p}$ ($\mathbf{q} = \mathbf{p}^* \cdot \mathbf{l}$), refract the direction as we did in section 4.2.5 where we used 3D GA, and construct the new line.

4.5 5D conformal model using geometric algebra

In the 5D conformal model [5] (which was called double homogeneous model in [1b]), two extra basis vectors are used, as opposed to one in the homogeneous model. One basis vector, $\mathbf{e}_0$, is used to represent the point at the origin, and the other, $\mathbf{e}_\infty$, is used to represent the point at infinity. These two basis vectors are reciprocal null vectors, which means:

$$\mathbf{e}_0 \cdot \mathbf{e}_0 = \mathbf{e}_\infty \cdot \mathbf{e}_\infty = 0 \tag{47}$$

$$\mathbf{e}_0 \cdot \mathbf{e}_\infty = 1 \tag{48}$$

Besides these two special basis vectors $\mathbf{e}_0$ and $\mathbf{e}_\infty$, there are three ordinary basis vectors $\mathbf{e}_1$, $\mathbf{e}_2$ and $\mathbf{e}_3$ that are equivalent to the traditional $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ axis.

This may seem a weird basis for a model of 3D Euclidean geometry, but you will see everything turns out nicely. If we consider the role of the extra basis vectors informally, we can motivate them as doing some extra administration of properties of our geometric objects, such that we can do many important geometric computations more easily.

Points are constructed as

$$\mathbf{q} = \vec{q} + \mathbf{e}_0 - \frac{1}{2}(\vec{q} \cdot \vec{q})\,\mathbf{e}_\infty \tag{49}$$

where $\vec{q}$ is a Euclidean 3D vector pointing from the origin to the location of the point $\mathbf{q}$.

Once we have defined our points, we don't need the origin $\mathbf{e}_0$ anymore and can construct extended objects (including lines, planes, point pairs, circles and spheres) by wedging the appropriate points together. To construct an object, one simply wedges together the appropriate set of characteristic primitives that is required to specify the object. E.g. a line $\mathbf{l}$ through the points $\mathbf{q}_1$ and $\mathbf{q}_2$ is constructed as:

$$\mathbf{l} = \mathbf{q}_1 \wedge \mathbf{q}_2 \wedge \mathbf{e}_\infty \tag{50}$$

Note the difference with the 4D GA model (equation 32): here we must also wedge $\mathbf{e}_\infty$. A plane can be constructed by wedging three points plus infinity. To construct a circle $\mathbf{c}$ through three points $\mathbf{q}_1$, $\mathbf{q}_2$ and $\mathbf{q}_3$ we construct the blade

$$\mathbf{c} = \mathbf{q}_1 \wedge \mathbf{q}_2 \wedge \mathbf{q}_3 \tag{51}$$

(so a line is actually a circle through infinity). If we now want to construct a sphere $\mathbf{s}$ that contains the circle $\mathbf{c}$ and a fourth point $\mathbf{q}_4$, we simply wedge them together

$$\mathbf{s} = \mathbf{c} \wedge \mathbf{q}_4 = \mathbf{q}_1 \wedge \mathbf{q}_2 \wedge \mathbf{q}_3 \wedge \mathbf{q}_4 \tag{52}$$

It is easy, straightforward and general to construct these objects. Since the outer product is anti-symmetric, 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 $\mathbf{q}_2 \wedge \mathbf{q}_1 \wedge \mathbf{e}_\infty$.

Rotors, used to represent rotations, can be constructed as the geometric product of vectors, or as exponents of bivectors. Since we have a point at infinity, $\mathbf{e}_\infty$, we can represent translations as 'rotations about infinity'. A translation over the vector $\vec{t}$ is represented by a translator

$$\mathbf{T} = \mathbf{1} + \frac{1}{2}\vec{t} \wedge \mathbf{e}_\infty = \mathbf{e}^{\frac{1}{2}\vec{t} \wedge \mathbf{e}_\infty} \tag{53}$$

This unites translations and rotations into a single versor representation. So if we first want to apply a translation represented by $\mathbf{T}$, followed by a rotation represented by $\mathbf{R}$, we compute the geometric product, $\mathbf{V} = \mathbf{R}\,\mathbf{T}$. This $\mathbf{V}$ can then be applied to any object. This is different from the 4D LA and 4D GA models where translations and rotations can only be unified by using transformation matrices or the outermorphism operator.

4.5.1 Rotation/translation

As explained above, a sequence of rotations and translations can be represented by a single versor. Any primitive $\mathbf{x}$ can be translated and rotated at the same time by applying the appropriate versor $\mathbf{V}$:

$$\mathbf{x}' = \mathbf{V}\,\mathbf{x}\,\mathbf{V}^{-1} \tag{54}$$

If translation and rotation are outermorphisms in 4D GA, then of course they are in 5D GA as well. So if we were to construct an outermorphism operator M from the versor $\mathbf{V}$, we could use

$$\mathbf{x}' = \mathbf{M}\mathbf{x} \tag{55}$$

instead of equation 54.

4.5.2 Line-plane intersection

To compute the intersection point $\mathbf{f}$ of a line $\mathbf{l}$ and a plane $\mathbf{p}$, we use the general equation for intersecting subspaces:

$$\mathbf{f} = \mathbf{p}^* \cdot \mathbf{l} \tag{56}$$

This construction (the inner product of one primitive and the dual of the other) can be used to compute the intersection of any pair of primitives. Even when the primitives do not intersect, the product will give a geometrically sensible answer that describes their *incidence* relationship.

Because the line and the plane intersect each other both at a point $\mathbf{q}$ and at infinity, $\mathbf{f}$ is a grade 2 object, a so-called *flat point*. This means that $\mathbf{f}$ is of the form

$$\mathbf{f} = \mathbf{q} \wedge \mathbf{e}_\infty \tag{57}$$

Although it is often possible to continue computations directly with $\mathbf{f}$, sometimes one would like to extract $\mathbf{q}$ from $\mathbf{f}$. Formally we can use the following for that purpose:

$$\begin{aligned} \mathbf{s}^* &= \mathbf{e}_0 \cdot \mathbf{f} && (58) \\ \mathbf{q} &= \mathbf{s}^* \, \mathbf{e}_\infty \, \mathbf{s}^{*^{-1}} && (59) \\ \mathbf{q}_n &= \frac{\mathbf{q}}{\mathbf{e}_\infty \cdot \mathbf{q}} && (60) \end{aligned}$$

In equation 58, 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 59, we reflect the point at infinity in the sphere to find its center point $\mathbf{q}$. In the next equation we normalize the point. In our ray tracer implementation however, $\mathbf{q}$ is more efficiently extracted from $\mathbf{f}$ by manipulating coordinates directly.

4.5.3 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:

$$\mathbf{r} = \mathbf{s}^* \cdot \mathbf{l} \tag{61}$$

We can check to see if the line and the sphere actually intersect by computing the radius ρ^4 of the 1D circle:

$$\rho^4 = \frac{\mathbf{r} \cdot \mathbf{r}}{4} \tag{62}$$

If ρ^4 is positive, the line and the sphere intersect. If ρ^4 is negative the line and the sphere don't intersect. If we need to, we can recover the two individual intersection points $\mathbf{q}_-$ and $\mathbf{q}_+$ from $\mathbf{r} = \mathbf{q}_- \wedge \mathbf{q}_+$ using this equation:

$$\mathbf{q}_\pm = \frac{\pm\sqrt{\mathbf{r} \cdot \mathbf{r}} + \mathbf{r}}{\mathbf{e}_\infty \cdot \mathbf{r}} \tag{63}$$

4.5.4 Reflection

Reflecting a line $\mathbf{l}$ in a plane $\mathbf{p}$ is done as follows:

$$\mathbf{l}' = \mathbf{p} \, \mathbf{l} \, \mathbf{p}^{-1} \tag{64}$$

This equation give 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 we had to in the other models.

4.5.5 Snell's law

Implementing Snell's law is quite straightforward in the conformal 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 runs through the intersection point of $\mathbf{l}$ and $\mathbf{p}$:

$$\mathbf{l}_n = \left(\frac{\mathbf{p}^* \cdot \mathbf{l}}{(\mathbf{e}_\infty \wedge \mathbf{e}_0) \cdot (\mathbf{p}^* \cdot \mathbf{l})} \cdot \mathbf{p} \right)^* \tag{65}$$

The refracted line is then computed as follows:

$$\mathbf{l}' = \left(\text{sign}(\mathbf{l} \cdot \mathbf{l}_n)\sqrt{1 - \eta^2 + (\mathbf{l} \cdot \mathbf{l}_n)^2\eta^2} - (\mathbf{l} \cdot \mathbf{l}_n)\eta\right)\mathbf{l}_n + \eta\mathbf{l} \tag{66}$$

Compare this to equation 12, a similar formula that merely computes the *directional* aspect, while here we work directly with lines in space.

5 Implementation and Performance

The ray tracers were implemented starting with the 5D GA model, simply because the authors were most curious about its performance. All other implementations were derived from that. We did not try 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.

5.1 Linear algebra implementation

The 3D and 4D linear algebra classes were implemented in an object oriented manner, taking efficiency into consideration. They do not use floating point SIMD instructions. Parts of the 4D Plücker coordinates code was taken directly from code generated by Gaigen, our own C++ GA package. But that code could just as easily have been copied from a textbook on projective geometry such as [10].

5.2 Geometric algebra implementation

The models that use geometric algebra were implemented using our own Gaigen. To the best of the authors knowledge, Gaigen is the most efficient geometric algebra implementation that is currently publicly available (at [3]).

Gaigen [2] is a program that can generate optimized C++ implementations of specific geometric algebras according to the user's specifications. It is the authors' first attempt at implementing GA efficiently. GA seems so general that it is very hard to write a single efficient implementation (e.g. a C++ template class) that implements every specific GA[2]. So what Gaigen can do is generate C++ source code for a specific GA for a specific application. Using Gaigen's user interface, the user specifies the properties of the GA required for the application at hand, and clicks a single button to generate the source code that implements that specific algebra. The properties of the algebra are things like name, dimensionality, signature, which products are required, which functions are required, optimizations, and how coordinates are stored.

Besides automatically generated code, another key idea behind Gaigen's efficiency is that it tracks the grade part usage of multivectors. Most objects we use in GA occupy only certain grade parts (a vector is always of grade 1, bivector is always of grade 2). Since grade part usage is known, Gaigen doesn't have to store the coordinates of 'empty' grade parts. This saves memory and

[2] Although http://jaap.flipcode.com/ga reports about such a GA implementation, using a technique called meta programming. This implementation is more efficient than Gaigen, but at the time of writing it was not mature enough to use it for the ray tracer benchmarks.

computation time, because no time is spent multiplying and adding values which are equal to zero anyway.

For even more efficiency, Gaigen allows you to add optimizations for specific products of specific objects. Imagine that the inner product of a 3-blade and a 2-blade is used 50% of the time in your application, you simply tell Gaigen to implement that product efficiently and regenerate the source code. To assist you in this optimization process, Gaigen can profile the application at run time and report which products should be optimized. This report can be read back into Gaigen's UI for automatic optimizations.

5.3 Performance

In figure 4 we present the benchmark results for each implementation of each model. There are two columns containing rendering times, one with and one without time spent on line-BSP intersection computations. This is because the full rendering time is dominated by computation of the intersection point of lines and polygonal models (stored in BSP trees). 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 makes use of only a few types of geometric computations. Thus we decided to add an (optional) pre-processing phase to the ray tracer. For every pixel, it 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. Being able to isolate the combinatorics of the intersection computations from the rest of the application, gets us two application benchmarks for the price of one. One application, the full ray tracing algorithm, spends its time mainly on line-BSP tree intersection tests. The other application performs a mix of all kinds of geometric computations.

As you can see in the table, 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, from platform to platform, from algorithm to algorithm.

6 Discussion, conclusion and future work

6.1 Discussion

Let us start with the conformal model. It is the clear winner in the elegance contest. All geometric primitives are directly represented using elements of the algebra. All geometric computations have been reduced to very elementary equations. In section 4.5.3 we showed that the conformal model allows us to use circles and spheres as direct elements of computation and we expect that this will have many advantages in other applications. A slight blemish are the two less elementary equations 58 and 63, which are used to extract points from bivectors. It may turn out that methods will be found to avoid these computations most of the time, but this is still an open issue.

model	implementation	full rendering time	rendering time w.o. BSP	executable size	run time memory usage
3D LA	standard	$1.00\times(23.3s)$	$1.00\times(0.99s)$	52KB	6.2MB
3D GA	Gaigen	$2.56\times$	$1.86\times$	64KB	6.7MB
4D LA	standard	$1.05\times$	$1.22\times$	56KB	6.4MB
4D GA	Gaigen	$2.97\times$	$2.62\times$	72KB	7.7MB
5D GA	Gaigen	$5.71\times$	$4.58\times$	100KB	9.9MB

Figure 4: Performance benchmarks run on a Pentium III 700 MHz notebook, with 256 MB memory, running Windows 2000. Programs were compiled using Visual C++ 6.0. All support libraries, such as fltk, libpng and libz were linked dynamically to get the executable size as small as possible. Run time memory usage was measured using the task manager.

Performance-wise, the conformal model is the big loser, being about $5\times$ slower than the most efficient models, and about $2\times$ slower than the other GA methods. This is partly due to some areas where Gaigen can be improved, and partly due to the model itself, which in some cases simply uses more computations or coordinates to do the same thing. 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 $\frac{32\times 32}{3\times 3} = 110\times$, achieving $5\times$ is not too bad. We are currently investigating methods to improve the performance of Gaigen in general and the conformal model specifically, which could be implemented in the next version of Gaigen. These include better data structures, coordinate usage tracking at the sub grade level and automatic simplicatication of expressions at the symbolical and coordinate levels. Single instruction multiple data (SIMD) instruction sets better fitted for the conformal model are also a possiblity, but it will probably be a while before general purpose CPUs are designed for doing geometric algebra efficiently.

By contrast, let us consider the most basic models, 3D LA and 3D GA. Judging by the equations alone, we see that in this particular application, 3D GA offers no great advantages over 3D LA. Sure, reflection equation 17 is nicer than 10, we can use and construct rotors (i.e. quaternions) more naturally, derive some equations more easily, but that's about it. However, some definite advantages will become obvious once one gets more familiar with GA. As discussed in [6], both the 3D LA and 4D LA models use the same vectors to represent a lot of different objects (directions, points, normal vectors, etc). The subtle differences in the way these vectors add and transform can lead to mysterious problems and hard to trace bugs. Switching to GA automatically resolves many of these problems. The grade mechanism of GA that allows for higher dimensional subspaces as direct elements of computation, lets us to make a distinction between objects that would otherwise appear the same, but act differently. A subjective advantage of GA that can not be uncovered by looking at the equations alone is the better understanding one might gain of geometry after learning GA. The authors benefited from this even while implementing the 3D LA and 4D LA models, for example in the derivation of equation 12 and

the use of Plücker coordinates.

When we look at the performance of the 3D models, we see that the 3D GA model using Gaigen is currently about $2\times$ 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 the Gaigen's soft typing of the geometric algebra objects at compile-time: all types of objects (scalars, vectors, bivectors, trivectors, rotors and so on) are represented by a single data type in Gaigen. When a product or operation has to be computed, Gaigen first 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 $5\times$ and $10\times$ 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 homogeneous model is most obvious in the construction of primitives, the use of the outermorphism for rotation/translation and the general intersection equation. Note that some of these improvements (like the outermorphism) could be used (and probably *are* used by some) in the 4D LA model. But due to the difficulty in understanding the 4D LA model such techniques are not as widespread as they should be. By studying GA one can add these techniques to the 4D LA model, in essence incorporating more of geometric algebra into the traditional model, which already contains elements that do not strictly belong there, like Plücker coordinates and quaternions. Unfortunately, due to deficiencies in the homogeneous model itself, other geometric computations, like reflection, are even more awkward to implement than in the 3D models. Most of these problems are resolved by the conformal model.

Performance of the 4D homogeneous model drops by a factor of about 3 due to Gaigen, but as said above, future GA implementations should reduce this to a small performance penalty, presumably less than $1.5\times$.

6.2 Conclusion

What can we conclude about this comparison? Mostly, that there is a sliding scale with performance on the one side and elegance on the other. For now, the traditional models (3D LA and 4D LA) remain most efficient and should be used in time critical applications.

For all other applications, like experiments, prototypes, one-time off-line tools and so on, we would like to recommend the elegant conformal model as the weapon of choice to tackle geometric problems. However, the conformal models isn't fully mature yet, although we don't expect this to take more than a couple of years. Currently there are no books, and few practical papers that describe the conformal model, but we and others are exploring the conformal model, theoretically, practically and educationally, to make it usable for the computer science community. So we recommend to study the conformal model now, keep an eye on developments, and possibly reap the benefits of that study 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 saw no great improvement in elegance of the 3D model 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 is a good source of new techniques and even code. In our implementation of the 4D LA model, we even directly copied code (fault free and automatically generated by Gaigen) from the 4D GA implementation to the 4D LA implementation. For application programmers, that may be the place of GA in their suite of techniques: to generate better LA code. But we expect that many will eventually program directly in GA.

7 Acknowledgements

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

References

[1a] L. Dorst and S. Mann. *Geometric algebra: a computation framework for geometrical application, Part 1.* IEEE Computer Graphics and Applications, Vol. 22, No. 3, pp 24-31, May/June 2002.

[1b] S. Mann and L. Dorst. *Geometric algebra: a computation framework for geometrical application, Part 2.* IEEE Computer Graphics and Applications, Vol. 22, No. 4, pp 58-67, July/August 2002.

[2] D. Fontijne, T. Bouma and L. Dorst. *Gaigen: a Geometric Algebra Implementation Generator.* Available at [3].

[3] D. Fontijne. *Gaigen and ray tracer website.* http://www.science.uva.nl/~fontijne/raytracer.

[4] A. S. Glassner (editor). *An Introduction To Ray Tracing.* Academic Press, 1989.

[5] D. Hestenes, H. Li, A. Rockwood. *A Unified Algebraic Framework for Classical Geometry.* In: G. Sommer (ed.), Geometric Computing with Clifford Algebra, Springer, Berlin, (1999). Also available from http://modelingnts.la.asu.edu/html/UAFCG.html.

[6] R. Goldman. *Illicit Expressions in Vector Algebra.* ACM Transactions on Graphics, Vol. 4, No. 3, July 1985.

[7] R. Goldman. *On the Algebraic and Geometric Foundations of Computer Graphics.* ACM Transaction of Graphics, Vol. 21, No.1, January 2002

[8] S. Mann, N. Litke, T. DeRose. *A Coordinate Free Geometry ADT.* University of Waterloo, Research Report CS-97-15.

[9] T. DeRose. *Coordinate-free geometric programming.* Technical Report 89-09-16, University of Washington, Department of Computer Science, Seattle, WA 98195 USA, September 1989.

[10] J. Stolfi. *Oriented Projective Geometry.* 1991, Academic Press.

A Derivation of 3D GA refraction equation

As promised, here we use 3D GA to derive equation 12. The interested reader might compare this with [4], which contains 2 two 3D LA derivations of the same equation. $\mathbf{u}$ is the direction of the incoming ray, $\mathbf{n}$ is the dual of the bivector $\mathbf{p}$ representing the plane, i.e. the normal vector. $\eta = \frac{\eta_1}{\eta_2}$ is a constant depending on the speed of light in both media. We want to compute $\mathbf{u}'$, the direction of the outgoing ray. In 3D GA, Snell's law can be fully specified by this set of equations:

$$\begin{aligned} \mathbf{u}' \wedge \mathbf{n} &= \eta \mathbf{u} \wedge \mathbf{n} && (67) \\ \mathbf{u}'^2 &= \mathbf{u}^2 && (68) \\ \text{sign}(\mathbf{u}' \cdot \mathbf{n}) &= \text{sign}(\mathbf{u} \cdot \mathbf{n}) && (69) \end{aligned}$$

Equation 67 states that $\mathbf{u}$, $\mathbf{u}'$ and $\mathbf{n}$ must all lie in the same plane, while the *sizes* of both bivectors are related by the constant η. Equation 68 simply states that the lengths of $\mathbf{u}'$ and $\mathbf{u}$ must be equal, while equation 69 states that $\mathbf{u}'$ and $\mathbf{u}$ must both have the same heading with respect to $\mathbf{n}$. We would like to extract $\mathbf{u}'$ from equation 67. The fact that the sum of the inner and outer product of $\mathbf{u}'$ and $\mathbf{n}$ is equal to their geometric product

$$\mathbf{u}'\mathbf{n} = \mathbf{u}' \cdot \mathbf{n} + \mathbf{u}' \wedge \mathbf{n} \tag{70}$$

suggests that, if we were able to express $\mathbf{u}' \cdot \mathbf{n}$ without using $\mathbf{u}'$, we could add $\mathbf{u}' \cdot \mathbf{n}$ to the LHS of to equation 67, divide ([1a], pg. 30) by $\mathbf{n}$, and get the answer. To find an expression for $\mathbf{u}' \cdot \mathbf{n}$, we note that

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

From this and equation 69 it follows that

$$\mathbf{u}' \cdot \mathbf{n} = \text{sign}(\mathbf{u} \cdot \mathbf{n})\sqrt{\mathbf{n}^2\mathbf{u}^2 + \eta^2(\mathbf{u} \wedge \mathbf{n})^2} \tag{71}$$

If we now add equation 71 to equation 67 we get:

$$\mathbf{u}' \wedge \mathbf{n} + \mathbf{u}' \cdot \mathbf{n} = \eta \mathbf{u} \wedge \mathbf{n} + \text{sign}(\mathbf{u} \cdot \mathbf{n})\sqrt{\mathbf{n}^2\mathbf{u}^2 + \eta^2(\mathbf{u} \wedge \mathbf{n})^2} \tag{72}$$

If we compare the LHS of this equation to the RHS of equation 70, we see that it is the (invertible) geometric product of $\mathbf{u}'$ and $\mathbf{n}$, so we divide by $\mathbf{n}$ and are done:

$$\mathbf{u}' = \frac{\eta \mathbf{u} \wedge \mathbf{n} + \text{sign}(\mathbf{u} \cdot \mathbf{n})\sqrt{\mathbf{n}^2\mathbf{u}^2 + \eta^2(\mathbf{u} \wedge \mathbf{n})^2}}{\mathbf{n}} \tag{73}$$

If both $\mathbf{n}$ and $\mathbf{u}$ have unit length we can simplify this to

$$\begin{aligned} \mathbf{u}' &= \eta(\mathbf{u} \wedge \mathbf{n})\mathbf{n} + \left(\text{sign}(\mathbf{u} \cdot \mathbf{n})\sqrt{1 + \eta^2(\mathbf{u} \wedge \mathbf{n})^2}\right)\mathbf{n} && (74) \\ &= \eta \mathbf{u} + \left(\text{sign}(\mathbf{u} \cdot \mathbf{n})\sqrt{1 + \eta^2(\mathbf{u} \wedge \mathbf{n})^2} - \eta \mathbf{u} \cdot \mathbf{n}\right)\mathbf{n} && (75) \end{aligned}$$

The last step that remains to derive 3D LA equation 12 is apply the fact

$$(\mathbf{u} \wedge \mathbf{n})^2 = (\mathbf{u} \cdot \mathbf{n})^2 - 1 \quad (76)$$

This is true because the magnitude of $(\mathbf{u} \wedge \mathbf{n})^2$ is equal to *minus* the square of the cosine of the angle between $\mathbf{u}$ and $\mathbf{n}$, and the magnitude of $(\mathbf{u} \cdot \mathbf{n})^2$ is equal to the square of sine of the angle between $\mathbf{u}$ and $\mathbf{n}$.

B What is available online

To accompany this paper we have constructed a web page [3] containing the following:

- a total of nine implementations of the ray tracing algorithm,
- ray tracing algorithm specification,
- more detailed benchmarks, comparing Gaigen to CLU, another C++ package,
- two tables summarizing all equations used in this paper,
- tutorials,
- Gaigen: a Geometric Algebra Implementation Generator. Papers, documentation and software
- links to other GA software and resources.