Computer Graphics using Conformal Geometric Algebra

Richard James Wareham
Robinson College

November 2006

A dissertation submitted to the
University of Cambridge
for the degree of
Doctor of Philosophy

Signal Processing and Communications Laboratory
Department of Engineering

UNIVERSITY OF CAMBRIDGE

This thesis contains no more than
umprint48550 words.

Typeset using the LaTeX document preparation system in the Sabon, Georgia and Computer Modern fonts.

SIGNAL PROCESSING AND COMMUNICATIONS LABORATORY,
Department of Engineering,
University of Cambridge,
Trumpington Street,
Cambridge, CB2 1PZ, U.K.

To my parents and
all those that helped
me get here

Contents

List of Figures

List of Tables

"A good Christian should beware that mathematicians, and any others who prophesy impiously... may be entangled in the companionship of demons."

— St Augustine

1

Introduction

It is generally accepted that a four-dimensional projective description of three-dimensional Euclidean geometry can have various advantages, particularly when intersections of planes and lines are required. Such projective descriptions are used extensively in computer vision and graphics where rotations and translations are usually described by a single 4×4 matrix. Since its inception in the mid-1970s, computer graphics (CG) has almost universally used linear vector algebra as its mathematical framework. This is due primarily to two factors: most early practitioners of computer graphics were mathematicians familiar with it; and linear algebra provided a compact, efficient way of representing points, transformations, lines, etc.

In the early 1980s CG moved out of the realm of Computer Science research and started to be used in the broader scientific community as an important research tool both for simulation and visualisation. Computer Graphics was then, and to an extent still is, tied to classical vector algebra which has started to show a number of flaws when applied to the problems being

investigated. Amongst these problems were: poor generalisation to spaces other than $\mathbb{R}^3$; great conceptual difficulty in extending problems to non-Euclidean geometries; and manipulating geometric objects other than simple lines, planes and points.

As computing power becomes cheaper, the opportunity arises to investigate new frameworks for CG which, although perhaps not providing the time and space efficiency of vector algebra, may provide a conceptually simpler system or one of greater analytical power. This thesis investigates the suitability of Geometric Algebra (GA) as one such approach. As one might hope, the original four-dimensional description of projective geometry fits very nicely into the Geometric Algebra framework[30, 36].

This thesis investigates the emerging field of *Conformal Geometric Algebra* (CGA) as a new basis for a CG framework. Computer Graphics is, fundamentally, a particular application of geometry. From a practical standpoint many of the low-level problems to do with rasterising triangles and projecting a three-dimensional world onto a computer screen have been solved and hardware especially designed for this task is available.

In the following chapters we start by assuming that good solutions to these problems are available and we investigate the use of CGA as a geometric building block sitting above the hardware. It is convenient that we may draw several million triangles in 3d space onto the screen but what if we wish to visualise a geometry other than Euclidean? Similarly we can draw several hundred characters onto the screen but this does not help us detect

intersections.

Consequently this work aims to investigate how CGA may help in these tangential CG problems and how problems that are not necessarily of traditional interest, e.g. the depiction of non-Euclidean geometries, may be tackled.

Within the thesis we tackle the following problems.

1.1 Non-Euclidean geometries

In chapter 5, we shall discuss and present a framework for visualising non-Euclidean geometries. We shall particularly emphasise hyperbolic geometry due to its novelty in the CG field and show how the treatment of the geometry in CG can relate to existing work[57] in representing spherical geometries.

1.2 Fractals

In chapter 6 we shall discuss an extension of complex number and their geometric analogue in hyperbolic space. As a simple application the generalisation of escape-time fractals to hyperbolic geometry will be shown.

1.3 Rotor exponentiation

In chapter 7 we shall extend some work[54] on the representation of rigid-body transforms via a 6 degree of freedom linear parameter space. Importantly, for application to existing systems, we show how we may map to

and from this representation and the 4×4 transformation matrices used in a number of existing systems.

The ability to map a description of rigid-body transformations into a linear space is immensely useful when attempting to smoothly interpolate pose and position. Similarly being able to extend many algorithms which work in a linear parameter space to position and pose allows a great many optimisation algorithms to be extended to cover rotation and translation naturally.

1.4 GPU-based techniques

In chapter 9 the techniques developed in chapter 7 are implemented on a commercial graphics processing unit (GPU). Simple mesh deformation and collision detection examples are shown. The examples aim to show that not only is GA a natural language for developing such algorithms that they are also compact enough to satisfy the space constraints of real consumer-grade hardware.

"You know we all became mathematicians for the same reason: we were lazy."
— Max Rosenlicht

2

An Overview of Geometric Algebra

In this chapter we present a brief overview of Geometric Algebra and how it may be used in a number of applications.

2.1 A Brief Overview of Geometric Algebra

As stated in the introduction, classical vector algebra has a number of problems once one moves away from three dimensional Euclidean space. Perhaps the clearest example is the cross-product of two vectors: for two vectors $\mathbf{a}, \mathbf{b}$ with lengths a, b, the product, $\mathbf{a} \times \mathbf{b}$, is conventionally defined as a vector normal to the plane containing $\mathbf{a}$ and $\mathbf{b}$ and has magnitude $ab \sin\theta$ where θ is the angle between $\mathbf{a}$ and $\mathbf{b}$. However the normal to the plane is only uniquely defined in three dimensions and has no meaning in 2- or 1-dimensional space; the cross-product does not generalise to higher-dimension spaces. The product is an important element of vector algebra and one can see that performing geometric operations in higher spaces without it quickly becomes complex.

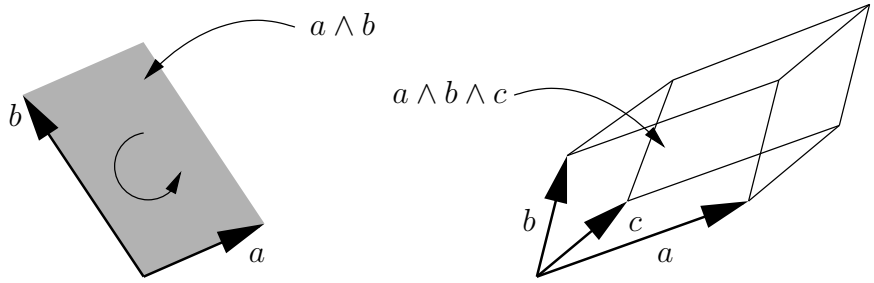


Figure 2.1: Illustration of bi- and trivectors

2.1.1 The products

It was in an attempt[25] to create an algebra of vectors which did generalise to higher spaces that a German schoolteacher named Hermann Graßmann (1809–1877) created an *exterior* or *outer* product of two vectors denoted as $a \wedge b$. For the remainder of this thesis we have dropped the usual convention of emboldening vectors since in GA they lie in the same algebra as scalars and do not need to be differentiated; the nature of the element is either stated explicitly or clear from context.

Graßmann's outer product is usually visualised geometrically as the movement of one vector along the other to form a 'directed area'. This is a new object, neither a vector nor a scalar. It is termed a *bivector*. Similarly one may form the outer product of this bivector with another vector to form a directed volume, a *trivector*, or generally a n-volume termed an n-vector.

To differentiate between scalars, vectors, bivectors, etc we say that a scalar is grade 0, a vector is grade 1, a bivector (formed from two vectors) is grade 2, etc. A n-vector has grade n.

The usual geometric visualisation is illustrated in figure 2.1. It is worth noting that other visualisations may be more appropriate for a specific application so the reader should not assume a bivector can *only* represent a directed area.

A key feature of GA is that the outer-product is anti-commutative and associative giving

$$a \wedge b = -(b \wedge a) \quad \text{and} \quad a \wedge (b \wedge c) = (a \wedge b) \wedge c = a \wedge b \wedge c.$$

Most of Graßmann's work was largely ignored by the mathematical community. It was not until William Clifford (1845–1879) investigated Graßmann's algebra in 1878[13] that the crucial step which made GA a useful algebra was made. Unfortunately Clifford's work was, in its turn, also somewhat ignored in favour of the contemporary work done by William Hamilton (1805–1865) on quaternions and the development of linear and vector algebra. In fact, as we shall see later, quaternions are simply a natural subset of the full Geometric Algebra over $\mathbb{R}^3$.

Clifford unified Graßmann's outer product and the familiar dot or inner-product into one *geometric product* such that

$$ab = a \cdot b + a \wedge b.$$

An algebra with this product is usually termed a *Clifford algebra*. We shall use the term Geometric Algebra to mean the *coupling* of Clifford algebras with an accompanying geometric interpretation.

A second glance at the geometric product shows an interesting feature which should be noted. In the expression above we are adding a scalar $(a \cdot b)$

to a bivector ($a \wedge b$). That is we are adding a grade 0 element to a grade 2 element. This combination of differing grade objects is analogous to complex numbers where we linearly combine a real and imaginary number to form a complex number. In this case we refer to such a combination of objects of varying grade as a *multivector*. We shall refer to all single-grade elements with lower-case letters but use upper-case letters to refer to multivectors. The geometric product also gives us a convenient new definition of the outer and inner products for vectors as

$$a \wedge b = \frac{1}{2}(ab - ba) \text{ and}$$

$$a \cdot b = \frac{1}{2}(ab + ba).$$

The power of this approach may be illustrated through its application to rotation. In two dimensions this is easily performed using complex numbers; representing the vector $[x\ y]$ as the complex number $z = x + iy$, rotation by θ radians can be performed by multiplication with $e^{i\theta}$. Hamilton worked for many years to extend this approach to three-dimensions. He eventually created *quaternions* [26, 27], an algebra with 4 basis elements $\{1, \mathbf{i}, \mathbf{j}, \mathbf{k}\}$ from which all elements are generated through linear combination. This algebra, although functional, lacked an obvious geometrical interpretation and again didn't generalise easily to higher-dimensions. We shall visit quaternions later and show how they relate to GA.

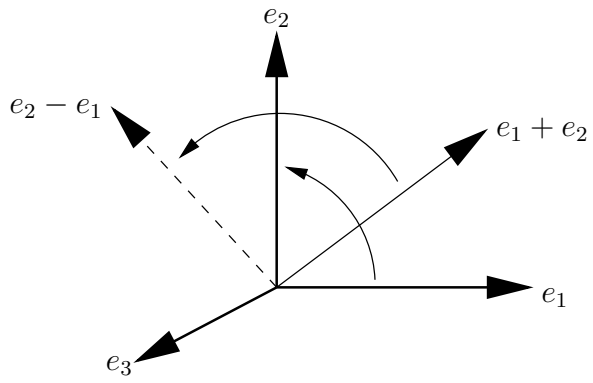


Figure 2.2: The rotation effect of bivector $B_3 = e_1 e_2$

2.1.2 Rotation via Rotors

To demonstrate Clifford's approach, consider any three orthonormal basis vectors of $\mathbb{R}^3$, $\{e_1, e_2, e_3\}$. We can form 3 different bivectors from these vectors:

$$B_1 = e_2 e_3, \quad B_2 = e_3 e_1, \quad B_3 = e_1 e_2.$$

Note that these are indeed bivectors since the basis vectors are orthogonal and

$$e_i e_j = e_i \cdot e_j + e_i \wedge e_j = e_i \wedge e_j \quad \text{iff} \quad i \neq j$$

Now consider the effect of B_3 on the vectors e_1 and $e_1 + e_2$:

$$e_1 B_3 = e_1^2 e_2 = e_2$$

$$(e_1 + e_2) B_3 = e_1 B_3 + e_2 B_3 = e_2 + e_2 e_1 e_2 = e_2 - e_1 e_2^2 = e_2 - e_1$$

From figure 2.2 it is clear that B_3 has the effect of rotating the vectors counter-clockwise by 90 degrees. It is, in fact, a general property that the bivector $e_i e_j$ will rotate a vector 90 degrees in the plane defined by e_i and e_j.

At first glance this seems to no more use than quaternions, but at no point have we assumed that we are working in three dimensional space – in fact this method also works in higher-dimension spaces.

We can also easily extend to general rotations; it is trivial to show that B_3 squares to -1:

$$B_3^2 = e_1 e_2 e_1 e_2 = -e_1 e_2 e_2 e_1 = -1$$

We can represent any vector x in the plane defined by e_1 and e_2 using

$$\begin{aligned} x &= r(e_1 \cos\theta + e_2 \sin\theta) \\ &= e_1 r(\cos\theta + B_3 \sin\theta) \end{aligned}$$

where r is the distance of the point x from the origin (i.e. $r = \sqrt{x^2}$) and θ is the angle x makes with e_1. By taking the Taylor expansion of cosine and sine and re-arranging the coefficients it can be shown that

$$e^{B_3\theta} = \cos\theta + B_3 \sin\theta$$

which is the GA analogue of de Moivre's theorem for complex numbers.

We can thus represent any vector x which lies in the plane of the bivector B_3 by

$$x = e_1 r e^{B_3\theta}$$

From this the same argument used for rotation in the complex plane can be used to show that rotation by ϕ radians in the plane of B_3 is accomplished by $x \mapsto x'$ where

$$x' = x e^{B_3\phi} = x(\cos\phi + B_3 \sin\phi)$$

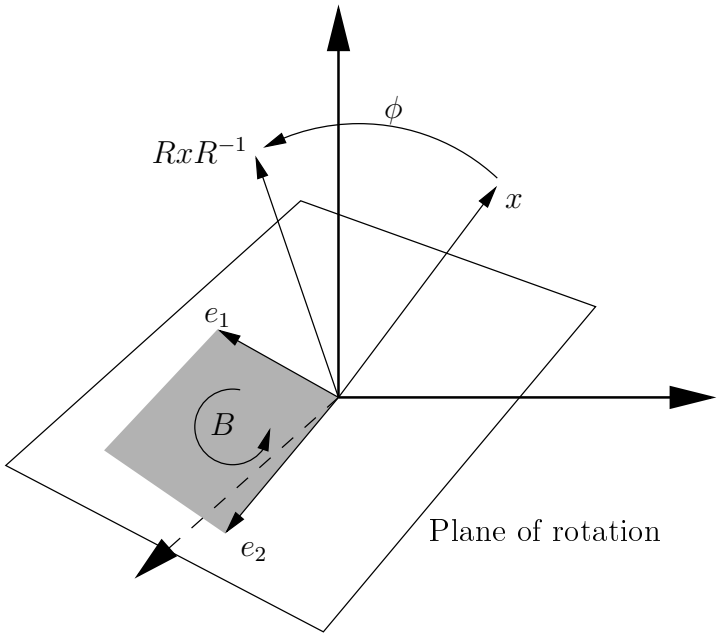


Figure 2.3: Rotating vectors in arbitrary planes

This has all taken place in two dimensions for the moment but nothing in our discussion has assumed this. In fact we can specify a unit bivector $B_3 = ab$ in three dimensions and rotate vectors in the place defined by a and b using the expression above.

Careful consideration must be given to the case where the vector to be rotated, x, does not lie on the plane of rotation as in figure 2.3. Firstly decompose the vector into a component which lies in the plane $x_{\parallel}$ and one normal to the plane $x_{\perp}$

$$x = x_{\parallel} + x_{\perp}$$

Now consider the effect of the following

$$\begin{aligned} e^{-B_3\phi/2} x e^{B_3\phi/2} &= \left(\cos\frac{\phi}{2} - B_3 \sin\frac{\phi}{2}\right)(x_\parallel + x_\perp)\left(\cos\frac{\phi}{2} + B_3\sin\frac{\phi}{2}\right) \\ &= x_\parallel(\cos\phi + B_3\sin\phi) + x_\perp \end{aligned}$$

since bivectors anti-commute with vectors in their plane (e.g. $e_1(e_2e_1) = -e_2 = -(e_2e_1)e_1$) and commute with vectors normal to the plane (e.g. $e_1(e_2e_3) = (e_2e_3)e_1$). We have thus succeeded in rotating the component of the vector which lies in the plane without affecting the component normal to the plane — we have rotated the vector around an axis normal to the plane.

This leads to a general method of rotation in any plane; we form a bivector of the form $R = \exp(-B\phi/2)$ for a given rotation ϕ in a plane specified by the unit bivector B. The transformation is therefore performed by

$$x \mapsto RxR^{-1}.$$

We refer to these bivectors which have a rotational effect as *rotors*. Figure 2.3 shows the various objects used. Later we shall extend the term *rotor* to refer to an element of the algebra which performs some well defined transformation.

Computing R^{-1} is rather difficult analytically and can sometimes require a full 2^n-dimension matrix inversion for a space of dimension n. To combat this we define the *reversion* of a n-vector $X = e_ie_j...e_k$ as

$$\tilde{X} = e_k...e_je_i$$

i.e. the literal reversion of the components. By looking at the expression for R it is clear that $\tilde{R} \equiv R^{-1}$ for rotors. Computing $\tilde{R}$ is easier since it generally

just involves changing the sign of components when an element is resolved onto a set of basis elements.

Note that in spaces with dimension n, the maximum grade object possible is an n-grade one. We denote the n-vector $e_1 \wedge \ldots \wedge e_n = I$ as the *pseudoscalar* and, for any element of the algebra, x, we term the product xI the *dual* of x represented as x^*. The dual is similarly defined for general multivectors. The pseudoscalar is so termed because it commutes with all elements of the algebra.

2.1.3 Relation to quaternions

It is worth comparing this method of rotation to rotation via quaternions. The three bivectors B_1, B_2 and B_3 act identically to the three 'imaginary' components of quaternions, $\mathbf{i}, -\mathbf{j}$ and $\mathbf{k}$ respectively. The sign difference between B_2 and $\mathbf{j}$ is due to the fact that the quaternions are not derived from the usual right-handed orthogonal co-ordinate system. This handedness mismatch often leads to annoying sign errors in quaternion-based algorithms.

Using quaternions, a particular rotation is represented via the unit quaternion q given by

$$q = q_0 + q_1\mathbf{i} + q_2\mathbf{j} + q_3\mathbf{k}$$

where $q_0^2 + q_1^2 + q_2^2 + q_3^2 = 1$. It is known[61] that quaternions are particularly useful not only for representing rotations but also for interpolating them. This interpolation is performed by considering the unit quaternion to be a point on a four-dimensional hyper-sphere and interpolating over the surface

of this sphere. For example, given a pair of rotations specified by the unit quaternions q_0 and q_1, the *spherical linear interpolation* (SLERP) would be

$$q = \begin{cases} q_0(q_0^{-1}q_1)^\lambda & \text{if } q_0 \cdot q_1 \geq 0 \\ q_0(q_0^{-1}(-q_1))^\lambda & \text{otherwise} \end{cases}$$

where λ varies in the range $(0,1)$ [39].

Recall that the locus of $\exp(i\theta)$ is the unit circle. It is straightforward to show that, for some normalised bivector B, the locus of the action of $\exp(B\theta)$ upon a point with respect to varying θ is also a circle in the plane of B. Hence, if we consider some rotations $R_1, R_2 = \exp(B)R_1$, where B is some normalised bivector, it is clear that the quaternionic interpolation is exactly given by

$$R_\lambda = (R_2\tilde{R}_1)^\lambda R_1 = \exp(\lambda B)R_1$$

where λ, the interpolation parameter, varies in the range $(0,1)$. In fact this method is not confined to three dimensions, like quaternionic interpolation, but instead readily generalises to higher-dimensions. It is also worth noting the extreme similarity between the rotor interpolation and the quaternionic interpolation. In fact pure-rotation rotors behave identically to quaternions and we shall see later how this SLERP interpolation scheme may be extended to include translation as well as rotation.

2.1.4 The Conformal Model of Geometric Algebra

In the Conformal Model[29] we extend the space by adding two additional basis vectors. The notation we use will follow the original notation given in

[29]. Let x be a vector in a space denoted $\mathcal{A}(p,q)$. The annotation (p,q) shall be termed the *signature* of the space. A given signature (p,q) implies that we may construct an orthogonal basis for the space, $\{e_i\}$, $i = 1,\cdots,n = p+q$ where $e_i^2 = +1$ for $i = 1,\cdots,p$ and $e_i^2 = -1$ for $i = p+1,\cdots,n$; i.e. we take a general mixed signature space.

To perform geometric operations using GA we now extend the space to $\mathcal{A}(p+1,q+1)$ via the inclusion of two additional orthogonal basis vectors, e and $\bar{e}$, such that

$$e^2 = +1, \quad \bar{e}^2 = -1$$

Note that if $x \in \mathcal{A}(p,q)$, then $e \cdot x = \bar{e} \cdot x = 0$ since $e_i \cdot e = e_i \cdot \bar{e} = 0$ for $i = 1,\cdots,n$. We now define vectors n and $\bar{n}$ as

$$n = e + \bar{e} \qquad \bar{n} = e - \bar{e}.$$

These vectors will be useful later. It is easy to see that n and $\bar{n}$ are *null* vectors since

$$\begin{aligned} n^2 &= (e+\bar{e}) \cdot (e+\bar{e}) = e^2 + 2e \cdot \bar{e} + \bar{e}^2 \\ &= 1 + 0 - 1 = 0 \end{aligned}$$

and

$$\begin{aligned} \bar{n}^2 &= (e-\bar{e}) \cdot (e-\bar{e}) = e^2 - 2e \cdot \bar{e} + \bar{e}^2 \\ &= 1 - 0 - 1 = 0. \end{aligned}$$

We also note two useful identities for n, $\bar{n}$ and $x \in \mathcal{A}(p,q)$:

$$\begin{aligned} n \cdot \bar{n} &= (e+\bar{e}) \cdot (e-\bar{e}) = e^2 - \bar{e}^2 = 2 \\ x \cdot n = x \cdot \bar{n} &= 0. \end{aligned}$$

It is equally easy to show that if we define the bivector $E = n \wedge \bar{n}$ then

$$\begin{aligned} E^2 &= (n \wedge \bar{n}) \cdot (n \wedge \bar{n}) \\ &= (n \cdot \bar{n})(n \cdot \bar{n}) - n^2 \bar{n}^2 \\ &= 4 \end{aligned} \tag{2.1}$$

since $n \cdot \bar{n} = 2$ and $n^2 = \bar{n}^2 = 0$.

In the conformal model we use general null-vectors to represent points and build up objects. In $\mathcal{A}(p,q)$ we map a point $x \in \mathcal{A}(p,q)$ to a vector $F(x) \in \mathcal{A}(p+1, q+1)$. Using this representation we find that complex geometric operations may be performed by simple algebraic manipulation of $F(x)$. The specific mapping used is the Hestenes ([29], page 302) representation

$$F(x) = -\frac{1}{2}(x-e)n(x-e) \tag{2.2}$$

where, substituting for $n = e + \bar{e}$ and using the fact that $\bar{e} \cdot x = 0 = \bar{e} \cdot e = n \cdot x$, it is possible to rewrite this equation in terms of the null vectors as follows

$$F(x) = \frac{1}{2}(x^2 n + 2x - \bar{n}) \tag{2.3}$$

which is similar to the form which is used in the more recent 'horosphere' formulations of the conformal framework [28]. We will see that there is some choice as to what factor we put in front of the $x^2 n + 2x - \bar{n}$ expression; we

choose $\frac{1}{2}$ so that our normalisation condition for null vectors, which allows us to compute the reverse mapping $F(x) \mapsto x$ independent of the absolute scale of $F(x)$, becomes

$$F(x) \cdot n = -1.$$

We will see that it will often be necessary to work with normalised lines, planes, circles and spheres specified by such 'unit' representations in order to apply the various formulæ which will be so important in later sections. It is also worth noting that the mapping as it stands above is dimensionally inconsistent. In following chapters we will show how compensating for this inconsistency can allow one to extend the approach to various non-Euclidean geometries.

It is possible to show that $F(x)$ is always a null vector for any x by directly evaluating $[F(x)]^2$

$$\begin{aligned} [F(x)]^2 &= \frac{1}{4}(x^2 n + 2x - \bar{n}) \cdot (x^2 n + 2x - \bar{n}) \\ &= -\frac{1}{2} x^2 n \cdot \bar{n} + x^2 \\ &= -x^2 + x^2 = 0 \end{aligned} \tag{2.4}$$

We have mapped vectors in $\mathcal{A}(p,q)$ into *null* vectors in $\mathcal{A}(p+1,q+1)$ and this is precisely the horosphere construction. It also shows the need to impose a normalisation constraint upon the resultant null vectors as they remain null irrespective of absolute scale. More generally we can show that all null vectors in $\mathcal{A}(p+1,q+1)$ must be the result of mapping some vector

$x \in \mathcal{A}(p,q)$ as above. Any vector $X \in \mathcal{A}(p+1,q+1)$ can be written as

$$X = an + bx + c\bar{n}$$

where $x = x^i e_i$, $i = 1, \cdots, p+q$ and hence $x \in \mathcal{A}(p,q)$. We can say that $X \cdot n = 2c$ and $X \cdot \bar{n} = 2a$ (since n is null and $x \cdot n = 0$). Therefore a and c are uniquely determined. However we can also write our general X as $X = bx^i e_i + \alpha e + \bar{\alpha}\bar{e}$ for suitable scalars α and $\bar{\alpha}$ and have $X \cdot e_j = bx^j$ $(j = 1, \cdots, p+q)$. So, whilst the product bx^j is uniquely defined by X, b and x^j individually are not. Now suppose that X is null so that $X^2 = 0$

$$\begin{aligned} X^2 &= (an + bx + c\bar{n}) \cdot (an + bx + c\bar{n}) \\ &= 2acn \cdot \bar{n} + b^2x^2 \\ &= b^2x^2 + 4ac = 0 \end{aligned} \tag{2.5}$$

From this we are easily able to see that any null vector can be written in the form

$$\lambda(x^2 n + 2x - \bar{n}) \tag{2.6}$$

since if $(an + bx + c\bar{n}) = \lambda(x^2 n + 2x - \bar{n})$ we have that

$$c = -\lambda \quad \lambda x^2 = a \quad \text{and} \quad 2\lambda = b$$

and we can then eliminate λ from these last two equations to give the condition $b^2x^2 = -4ac$. This is precisely the condition given in equation 2.5.

These results may now be used to provide a projective mapping between $\mathcal{A}(p,q)$ and $\mathcal{A}(p+1,q+1)$. Specifically that the family of null vectors $\lambda(x^2 n + 2x - \bar{n})$, in $\mathcal{A}(p+1,q+1)$ are taken to correspond to the single point $x \in \mathcal{A}(p,q)$.

If x is the origin then we see that $F(x) = -\frac{1}{2}\bar{n}$ and we may therefore associate null vectors parallel to $\bar{n}$ with the origin. We will see later that when we *invert* $\bar{n}$ we obtain n, suggesting that we associate null vectors parallel to n with the *point at infinity* (the usual result of inverting the origin).

When we look at the inner product of normalised null vectors in $\mathcal{A}(p+1, q+1)$ we discover something very interesting. If A and B in $\mathcal{A}(p+1, q+1)$ represent the points a and b in $\mathcal{A}(p,q)$, then

$$\begin{aligned} A \cdot B &= F(a) \cdot F(b) \\ &= \frac{1}{4}(a^2 n + 2a - \bar{n}) \cdot (b^2 n + 2b - \bar{n}) \\ &= -\frac{1}{2}a^2 + a \cdot b - \frac{1}{2}b^2 \\ &= -\frac{1}{2}(a-b)^2 \end{aligned} \tag{2.7}$$

and we see that $A \cdot B$ is related to the Euclidean distance between points a and b. We can therefore define the Euclidean distance between two point representations as

$$d(A,B) = \sqrt{-2(A \cdot B)} \tag{2.8}$$

This property of the conformal space and its relationship to *distance geometry* [17] is discussed at more length in [28]. The mapping and properties described here were outlined originally in [29].

Rotations

In usual descriptions of GA (without the use of the conformal model) rotations are performed with elements of the algebra termed rotors. In this

section we aim to show that these rotors may be used unchanged on the null vectors representing points. Let $x \mapsto Rx\tilde{R}$ with $x \in \mathcal{A}(p,q)$ and R be a rotor in the Geometric Algebra over $\mathcal{A}(p,q)$. Consider what happens when R acts upon $F(x)$; i.e. the nature of $RF(x)\tilde{R}$

$$RF(x)\tilde{R} = \frac{1}{2}R(x^2 n + 2x - \bar{n})\tilde{R} = \frac{1}{2}[x^2 Rn\tilde{R} + 2Rx\tilde{R} - R\bar{n}\tilde{R}].$$

It is straightforward to show that this commutes with n and $\bar{n}$ so

$$RF(x)\tilde{R} = \frac{1}{2}(\hat{x}^2 n + 2\hat{x} - \bar{n}) \tag{2.9}$$

where $\hat{x} = Rx\tilde{R}$. We have therefore shown that rotors in $\mathcal{A}(p,q)$ remain rotors in $\mathcal{A}(p+1,q+1)$ in that they retain their action about the point x represented by $F(x)$. To summarise

$$x \mapsto Rx\tilde{R} \qquad \Leftrightarrow \qquad F(x) \mapsto F(Rx\tilde{R}) \tag{2.10}$$

Translators

Translation along a vector a is defined for our purposes as the mapping $x \mapsto x+a$ for some $x, a \in \mathcal{A}(p,q)$. In this section we will show that this is performed by applying a rotor $R = T_a = \exp\left(\frac{na}{2}\right)$ to $F(x)$.

Before proceeding with the proof we should note that in some non-Euclidean geometries the addition operator, when viewed as a translation operator, is non-commutative; translation of geodesic A along geodesic B is not the same, in general, as translation of B along A. For the moment we shall ignore this but it will become important again in later chapters on non-Euclidean geometries.

Returning to the form of the translation rotor, consider the usual power series expansion of the exponential which we may immediately simplify to

$$R = T_a = \exp\left(\frac{na}{2}\right) = 1 + \frac{na}{2} + \frac{1}{2}\left(\frac{na}{2}\right)^2 + \cdots = 1 + \frac{na}{2} \tag{2.11}$$

since n is null, $an = -na$ and therefore the higher order terms are all zero. We now see how R acts on the vectors n, $\bar{n}$ and x.

$$\begin{aligned} Rn\tilde{R} &= \left(1 + \frac{na}{2}\right) n \left(1 + \frac{an}{2}\right) \\ &= n + \frac{1}{2}nan + \frac{1}{2}nan + \frac{1}{4}nanan \\ &= n \end{aligned} \tag{2.12}$$

again using $an = -na$ and $n^2 = 0$. Similarly we can show that

$$R\bar{n}\tilde{R} = \bar{n} - 2a - a^2 n \tag{2.13}$$

$$Rx\tilde{R} = x + n(a \cdot x) \tag{2.14}$$

Immediately we see that our interpretation of n being the point at infinity and $\bar{n}$ being the origin is consistent with our claim that R represents the translation $x \mapsto x + a$ since $R(-\bar{n})\tilde{R} = F(a)$ and the point at infinity is unchanged by finite translation.

We can now also see how the rotor acts on $F(x)$

$$\begin{aligned} RF(x)\tilde{R} &= \left(1 + \frac{na}{2}\right) \frac{1}{2}(x^2 n + 2x - \bar{n}) \left(1 + \frac{an}{2}\right) \\ &= \frac{1}{2}(x^2 n + 2(x + n(a \cdot x)) - (\bar{n} - 2a - a^2 n)) \\ &= \frac{1}{2}((x + a)^2 n + 2(x + a) - \bar{n}) \\ &= \frac{1}{2}(\hat{x}^2 n + 2\hat{x} - \bar{n}) = F(x + a) \end{aligned} \tag{2.15}$$

where $\hat{x} = x + a$ and thus translations in $\mathcal{A}(p,q)$ can be performed by the rotor $R = T_a$ defined above. To summarise

$$x \mapsto x + a \qquad \Leftrightarrow \qquad F(x) \mapsto T_a F(x) \tilde{T}_a = F(x + a) \tag{2.16}$$

Inversion

In the usual three-dimensional geometric algebra we can reflect a vector a in a plane with unit normal n by 'sandwiching' the vector between the normal $-nan$ [37]. Sandwiching the object *to be reflected* between the object *in which we wish to reflect* is a very general prescription in GA and one which will be used heavily in later parts of this thesis. In this section we look at how inversions are brought about by this same reflection operation.

By 'inversion' we mean the mapping $x \mapsto \frac{x}{x^2}$ or, equivalently, for non-singular vectors, $x \mapsto x^{-1}$. Firstly, we look at the reflection in e of various vectors

$$-ene = -ee\bar{n} = -\bar{n}$$

since $ne = (e + \bar{e})e = (e^2 + \bar{e}e) = (e^2 - e\bar{e}) = e\bar{n}$. Similarly, we can show that a number of reflection properties hold

$$-ene = -\bar{n} \tag{2.17}$$

$$-e\bar{n}e = -n \tag{2.18}$$

$$-exe = x \tag{2.19}$$

and finally we may observe what happens to $F(x)$ under reflection in e

$$\begin{aligned} -eF(x)e &= -e\frac{1}{2}(x^2n+2x-\bar{n})e \\ &= \frac{1}{2}[-x^2\bar{n}+2x+n] \\ &= x^2\frac{1}{2}\left[\frac{1}{x^2}n+2\frac{x}{x^2}-\bar{n}\right] \\ &= x^2F\left(\frac{x}{x^2}\right) \end{aligned} \tag{2.20}$$

We have, therefore, shown that the inversion operation in $\mathcal{A}(p,q)$ can be performed via the reflection in e of the representation in $\mathcal{A}(p+1,q+1)$.

$$x \mapsto \frac{x}{x^2} \qquad \Leftrightarrow \qquad F(x) \mapsto -\frac{eF(x)e}{x^2} = F\left(\frac{x}{x^2}\right) \tag{2.21}$$

Since the absolute scale of $F(x)$ is irrelevant, as we always rescale to impose our normalisation constraint, we can omit the scaling by x^{-2}. It is also irrelevant, by the same logic, whether we take $-e(\cdot)e$ or $e(\cdot)e$ as the reflection and henceforth we will use $e(\cdot)e$ for convenience. This sandwiching operation will be a common one in the algorithms we will describe in subsequent chapters.

Dilators

A dilation by a factor of α is represented by the mapping $x \mapsto \alpha x$. In this section we investigate how to form a rotor which has the action of dilating about the origin. We start by considering the rotor $R = D_\alpha = \exp\left(\frac{\alpha}{2}e\bar{e}\right)$ and a number of relations which can easily be verified

$$\begin{aligned} -e\bar{e}n &= n &&= ne\bar{e} \\ -\bar{n}e\bar{e} &= \bar{n} &&= e\bar{e}\bar{n} \end{aligned} \tag{2.22}$$

We can now look at what $RF(x)\tilde{R}$ gives

$$\begin{aligned} D_\alpha F(x)\tilde{D}_\alpha &= \exp\left(\frac{\alpha}{2}e\bar{e}\right)\frac{1}{2}\{x^2 n + 2x - \bar{n}\}\exp\left(-\frac{\alpha}{2}e\bar{e}\right) \\ &= \frac{1}{2}(x^2\exp(\alpha e\bar{e})\,n + 2x - \exp(\alpha e\bar{e})\,\bar{n}) \\ &= \frac{1}{2}(x^2\exp(-\alpha)\,n + 2x - \exp(\alpha)\,\bar{n}) \\ &= \exp(\alpha)\frac{1}{2}\{\exp(-2\alpha)\,x^2 n + 2\exp(-\alpha)\,x - \bar{n}\} \\ &= \exp(\alpha)\frac{1}{2}\{\hat{x}^2 n + 2\hat{x} - \bar{n}\} \end{aligned} \tag{2.23}$$

where $\hat{x} = \exp(-\alpha)\,x$. The above steps can be verified by considering $\exp\left(-\frac{\alpha}{2}e\bar{e}\right)$ as the expansion $1 - \frac{\alpha}{2}e\bar{e} + \frac{1}{2!}\left(\frac{\alpha}{2}e\bar{e}\right)^2 + \cdots$ and using the relations given in equation 2.22. Again noting that the absolute scale of $F(x)$ is unimportant we have therefore shown that dilations by a factor of $\exp(-\alpha)$ can be performed by the rotor $R = D_\alpha$

$$x \mapsto \exp(-\alpha)x \qquad \Leftrightarrow \qquad F(x) \mapsto D_\alpha F(x)\tilde{D}_\alpha = \exp(\alpha)F(\exp(-\alpha)x) \tag{2.24}$$

Note that the signs are incorrect in the equivalent equations in [29], p.303, equation 3.22. It is worth noting that dilation about any other point may be achieved by concatenating the appropriate rotors to move that point to the origin, dilate and move back.

Special conformal transforms

We have seen above that we are able to express rotations, inversions, translations and dilations in $\mathcal{A}(p,q)$ by rotations and reflections in $\mathcal{A}(p+1,q+1)$. This now leads us to consider *special conformal transformations*. These are es-

sentially transformations which preserve angles and are defined by the motion

$$x \mapsto x\frac{1}{1+ax} \tag{2.25}$$

Some thought reveals this transform to be a combination of inversion, translation and inversion again

$$\begin{aligned} x \quad &\overset{\text{inversion}}{\longmapsto} \quad \frac{x}{x^2} \\ &\overset{\text{translation}}{\longmapsto} \quad \frac{x}{x^2}+a \equiv \frac{x}{x^2}(1+xa) \\ &\overset{\text{inversion}}{\longmapsto} \quad \frac{\frac{x}{x^2}+a}{(\frac{x}{x^2}+a)(\frac{x}{x^2}+a)} \\ &= \quad \frac{x+ax^2}{1+2a\cdot x+a^2x^2} = x\frac{1}{1+ax} \end{aligned} \tag{2.26}$$

since $\frac{1}{1+ax} = \frac{1+xa}{(1+ax)(1+xa)}$. The final line in the above expression shows us that $x\frac{1}{1+ax}$ is indeed a vector since $x+ax^2$ is a vector. As we have built up the special conformal transformation via inversions and translations, we know exactly how to construct the $\mathcal{A}(p+1,q+1)$ operator that performs such a transformation by simply chaining the rotors for inversion and translation we derived above. The required rotor is therefore given by

$$K_a = eT_ae, \qquad \text{so that} \qquad x \mapsto K_axK_a \tag{2.27}$$

and

$$K_axK_a = e\left\{T_a(exe)\tilde{T}_a\right\}e \tag{2.28}$$

Substituting for the rotors above we can write our special conformal rotor as

$$K_a = 1 - \frac{1}{2}\bar{n}a \tag{2.29}$$

We are now in a position to see what happens when we act on $F(x)$ with K_a

$$\begin{aligned} K_a F(x)\tilde{K}_a &= eT_a(eF(x)e)\tilde{T}_a e \\ &= eT_a(-x^2 F(\frac{x}{x^2}))\tilde{T}_a e \\ &= -x^2 e\left\{F(\frac{x}{x^2}+a)\right\}e \\ &= -x^2\left\{-\left(\frac{x}{x^2}+a\right)^2 F\left(\frac{\left(\frac{x}{x^2}+a\right)}{\left(\frac{x}{x^2}+a\right)^2}\right)\right\} \\ &= (1+2a\cdot x+a^2x^2)F\left(x\frac{1}{1+ax}\right) \end{aligned} \tag{2.30}$$

The end result is therefore

$$x \mapsto x\frac{1}{1+ax} \qquad \Leftrightarrow \qquad F(x) \mapsto (1+2a\cdot x+a^2x^2)F\left(x\frac{1}{1+ax}\right) \tag{2.31}$$

2.1.5 Observations

We can see that, from the above, the following results are true:

$$\begin{aligned} RnR &= n \quad \text{for } R \text{ a rotation, since } \; n\tilde{R} = \tilde{R}n \\ RnR &= n \quad \text{for } R \text{ a translation (equation 2.12)} \\ n\tilde{R} = \tilde{R}n \text{ and } \; RnR &= \exp(-\alpha)n \quad \text{for dilations} \end{aligned}$$

Thus rotations, translations and dilations leave n, which we identify with the point at infinity, unchanged up to a scale factor. This is a fact which will be important to us in subsequent chapters. Indeed, we find that the underlying geometry described by the rotors is related to the element of the algebra which the rotors hold invariant. We will see later that the five-dimensional conformal setup provides a framework in which we can simply describe non-Euclidean geometries in such terms.

2.2 Existing implementations

As part of the research presented here a library designed to help with the implementation of CGA-based algorithms in an efficient manner was created. Before work started on designing the software, several existing systems were investigated. All of the following packages were designed to provide high-level access to numerical computations using GA.

2.2.1 CLUCalc & CLUDraw

CLU & CLUDraw were written by Christian Perwass and may be obtained from his web-site[50]. Of all the systems, this is the only one designed both for CGA and the visualisation of spheres, circles, etc. directly from the CGA model.

It is written in C++ and uses the object-oriented features of the language extensively. Multivectors are represented as objects and operations upon them are performed by overloading the standard operators of the C++ language.

CLU is a library designed for numerical computations and is not limited to the signature used for the conformal model but also has support for other signatures. CLUDraw is a library designed to take multivectors calculated by CLU and to provide a convenient way to visualise them as spheres, lines, planes, etc.

Although it provides a convenient interface, the heavy use of C++ object-orientation and operator overloading within the library results in a rather

high computational overhead. The decoupling of the calculation engine and visualisation engine, however, provides the useful ability to isolate the graphics code in a clean manner.

2.2.2 Gaigen

The Gaigen homepage[22] describes it thus

> Gaigen is a program which can generate implementations of geometric algebras. It generates C++, C and assembly source code which implements a geometric algebra requested by the user. People who are new to geometric algebra may think that there is only one geometric algebra. However, there are many different geometric algebras. The properties that make these algebras different are, among others, their dimensionality and the signature of their basis vectors. Each of these different algebras may be useful for different applications.

The user can select the signature of the space and generate C-code to implement it. The code it generates is efficient and compact. The Gaigen2 project has moved into a different direction – instead of accelerating individual products, a different approach to making an efficient implementation is used with a different internal representation. Gaigen does not possess a visualisation engine by default.

Gaigen, although fine for general purpose use, was not suitable for this PhD as, ultimately, it was desired that a similar API would be useful for both

Geometric Product Multiplication Table

	1	e_1	e_2	e_3	e_{12}	e_{13}	e_{23}	e_{123}
1	$+1$	$+e_1$	$+e_2$	$+e_3$	$+e_{12}$	$+e_{13}$	$+e_{23}$	$+e_{123}$
e_1	$+e_1$	$+1$	$+e_{12}$	$+e_{13}$	$+e_2$	$+e_3$	$+e_{123}$	$+e_{23}$
e_2	$+e_2$	$-e_{12}$	$+1$	$+e_{23}$	$-e_1$	$-e_{123}$	$+e_3$	$-e_{13}$
e_3	$+e_3$	$-e_{13}$	$-e_{23}$	$+1$	$+e_{123}$	$-e_1$	$-e_2$	$+e_{12}$
e_{12}	$+e_{12}$	$-e_2$	$+e_1$	$+e_{123}$	-1	$-e_{23}$	$+e_{13}$	$-e_3$
e_{13}	$+e_{13}$	$-e_3$	$-e_{123}$	$+e_1$	$+e_{23}$	-1	$-e_{12}$	$+e_2$
e_{23}	$+e_{23}$	$+e_{123}$	$-e_3$	$+e_2$	$-e_{13}$	$+e_{12}$	-1	$-e_1$
e_{123}	$+e_{123}$	$+e_{23}$	$-e_{13}$	$+e_{12}$	$-e_3$	$+e_2$	$-e_1$	-1

Table 2.1: Example TEX output from Gaigen

software and hardware implementations. It was felt that having our own implementation would lead to greater flexibility in this area.

Gaigen does have the useful ability to generate product tables for the algebra (see table 2.1) in both plain text and TEX format.

2.2.3 Cambridge GA library for Maple

This library[10] provides Geometric Algebra capabilities for the Maple V and VI symbolic mathematics packages. It provides no visualisation capabilities above those provided by Maple. This is a very useful tool for research but is aimed more at symbolic manipulation than numerical computation.

2.3 Existing uses

A number of existing applications of GA have been developed in the field of graphics and vision[62, 23] and indeed much of contemporary physics has been recast using GA methods[15], providing a conceptually simpler framework for further research.

This thesis will build on such work and aims to advance, using Geometric Algebra, a number of fields in Computer Graphics.

"A topologist is one who doesn't know the difference between a doughnut and a coffee cup."

— John Kelley

3

Objects in the Conformal Representation

In this chapter we will look at the rôle played by bivectors in the conformal model and give some useful alternative representations of lines, planes, circles and spheres.

Throughout this chapter we shall say that a geometric primitive is represented by some element M of the algebra if, for all null-vector representations X which satisfy

$$X \wedge M = 0,$$

the point represented by X lies on the object or primitive and the representaion of all points on the object satisfy that relation. We term this an *incidence relation*.

Much of this work relies on the approach introduced by Rosenhahn[54] on representing objects via incidence relations. This chapter presents these result in a form which will be used later on along with a set of proofs showing the ease with which intrinsically geometric results can be shown with GA.

3.1 A note on methodology

In this section we will develop a useful tool used by a number of proofs below. Firstly recall that, in projective geometry, if a line, L, passes through two points a, b, whose (4d) homogeneous representations are A,B[1], we can represent the line by the bivector $L = A \wedge B$. The representation, X, of any point lying on the line will satisfy

$$X \wedge L = 0.$$

In the conformal representation rotations, translations, dilations and inversions are all represented by rotors or reflections, which allows us to infer that any incidence relations remain invariant in form under such operations. This may also be seen explicitly – suppose we have the incidence relation

$$X \wedge Y \wedge \cdots \wedge Z = 0$$

where $X, Y, \cdots, Z \in \mathcal{A}(p+1, q+1)$. Under reflections in e we have

$$\begin{aligned} X \wedge Y \wedge \cdots \wedge Z &\mapsto (eXe) \wedge (eYe) \wedge \cdots \wedge (eZe) \\ &= e(X \wedge Y \wedge \cdots \wedge Z)e \end{aligned}$$

since $(eXe) \wedge (eYe) = \frac{1}{2}(eXeeYe - eYeeXe) = \frac{1}{2}e(XY - YX)e = e(X \wedge Y)e$, as $e^2 = 1$. Therefore it is true that if $X \wedge Y \wedge \cdots \wedge Z = 0$ then $(eXe) \wedge (eYe) \wedge \cdots \wedge (eZe) = 0$.

Similarly, if we consider the relation under some rotor R we have

$$\begin{aligned} X \wedge Y \wedge \cdots \wedge Z &\mapsto (RX\tilde{R}) \wedge (RY\tilde{R}) \wedge \cdots \wedge (RZ\tilde{R}) \\ &= R(X \wedge Y \wedge \cdots \wedge Z)\tilde{R} \end{aligned}$$

[1] i.e. we add a further orthogonal basis vector e_0 so that $A = a + e_0$.

again using $(RX\tilde{R})\wedge(RY\tilde{R}) = \frac{1}{2}(RX\tilde{R}RY\tilde{R} - RY\tilde{R}RX\tilde{R}) = \frac{1}{2}R(XY - YX)\tilde{R} = R(X\wedge Y)\tilde{R}$, as $R\tilde{R} = 1$. Once more we can say that if $X\wedge Y\wedge\cdots\wedge Z = 0$ then it is also true that $(RX\tilde{R})\wedge(RY\tilde{R})\wedge\cdots\wedge(RZ\tilde{R}) = 0$.

Translations, rotations, dilations and inversions can now be brought into the context of projective geometry, giving a significant increase in the usefulness of the representation. It is now possible to build up a set of useful results in this conformal system and to see how lines, planes, circles and spheres are represented. The working above will be the basis for most proofs of constructions in this chapter. We prove the construction holds for some simple case at the origin and then we can say it must hold for all cases since we can rotate, translate and dilate our setup at the origin to any configuration in space, with no change in the incidence relations.

3.2 The equation of a line

As was discussed above, the incidence relations are invariant under rotations and translations in the $\mathcal{A}(p,q)$ space. Hence without loss of generality we can consider the incidence relation for a line in the direction e_1 passing through the origin.

Let three points on this line be x_1, x_2, x_3 with corresponding $\mathcal{A}(p+1, q+1)$ representations X_1, X_2, X_3. It is clear that $\{X_i\}$ contains only the vectors $n, \bar{n}$ and e_1 as any point x on the line must have the form $x = \lambda e_1$. We therefore claim that, if X is the representation of any other point on the line, we can write the

incidence relation

$$X \wedge X_1 \wedge X_2 \wedge X_3 = 0$$

We prove this as follows. Let $L = X_1 \wedge X_2 \wedge X_3$, if we expand this out in terms of the conformal representation we have

$$\begin{aligned} L &= \frac{1}{8}(x_1^2 n + 2x_1 - \bar{n}) \wedge (x_2^2 n + 2x_2 - \bar{n}) \wedge (x_3^2 n + 2x_3 - \bar{n}) \\ &= \alpha(n \wedge e_1 \wedge \bar{n}) \end{aligned} \tag{3.1}$$

where α is a scalar which depends upon x_1, x_2, x_3. If $X = \frac{1}{2}(x^2 n + 2\lambda e_1 - \bar{n})$, then it is easy to see by direct substitution that $X \wedge L = 0$. Since we can rotate and translate our simple line through the origin to any other line in space and still preserve the incidence relations, we know that we may represent all lines in this manner. It is interesting to note that this parallels the projective case and also that we would appear to require 3 points in this conformal representation to describe a line instead of the usual 2. We shall return to this later.

3.3 The equation of a plane

This section uses a similar method to develop the representation of a plane. Again by translational and rotational invariance we can, without loss of generality, consider initially only the plane spanned by e_1 and e_2 and passing through the origin. If the point x lies in this plane then we can write

$$x = \lambda e_1 + \mu e_2$$

and its conformal representation X will only contain the vectors $n, \bar{n}, e_1, e_2$

$$X = \frac{1}{2}(x^2 n + 2(\lambda e_1 + \mu e_2) - \bar{n})$$

Let $\Phi = X_1 \wedge X_2 \wedge X_3 \wedge X_4$, where the points represented by $\{X_i\}$ all lie in the plane. By expanding we can show that Φ must take the form

$$\Phi = \beta(n \wedge \bar{n} \wedge e_1 \wedge e_2)$$

and so, for any representation X of a point on the plane we have $X \wedge \Phi = 0$. Therefore

$$X \wedge X_1 \wedge X_2 \wedge X_3 \wedge X_4 = 0 \tag{3.2}$$

is the incidence relation for a plane passing through points represented by X_i, $i = 1, ..., 4$. Once again we note the apparent requirement for 4 points to specify the plane as opposed to the usual 3.

We may easily extend this to higher dimensions and specify an r-d hyperplane (where a line is $r = 1$, a plane is $r = 2$ and so forth) via the relation

$$X \wedge X_1 \wedge X_2 \wedge \ldots X_{r+1} \wedge X_{r+2} = 0 \tag{3.3}$$

where $\{X_i\}$ are conformal representations of the $r + 2$ points $\{x_i\}$ lying on the hyperplane.

3.4 The role of inversion: lines and circles

It may initially appear incongruous to specify $r + 2$ points in order to determine an r-d hyperplane. For example, 2 points clearly suffice to determine a

line, 3 points for a plane and so on. This section discusses the rôle of these extra points.

To understand the requirement for these extra points, and the part inversion plays, we shall use a simple example. We shall consider the space $\mathcal{A}(2,0)$, that is, the ordinary Euclidean plane with basis $\{e_1, e_2\}$, $e_1^2 = 1,\ \ e_2^2 = 1$.

Let the line L be $x = 1$, or equivalently, $(1,y) : -\infty \leq y \leq +\infty$ and let a be the point (x,y). Suppose we wish to invert points on this line. Doing so gives us the set of points

$$a \mapsto \frac{a}{a^2} \qquad \Longrightarrow \qquad L \mapsto \left(\frac{1}{1+y^2}, \frac{y}{1+y^2} \right) \tag{3.4}$$

Parameterising the original line as $x = 1, y = t;\ \ -\infty \leq t \leq +\infty$, the inversion produces $(x',y') = \left(\frac{1}{1+t^2}, \frac{t}{1+t^2}\right)$ and it is then easy to show that

$$\left[x' - \frac{1}{2}\right]^2 + y'^2 = \left(\frac{1}{2}\right)^2$$

The inversion, therefore, produces a circle through the origin, centre $(\frac{1}{2}, 0)$ radius $\frac{1}{2}$, see figure 3.1. We may therefore make the connection

$$\text{straight line} \qquad \overset{\text{inversion}}{\longmapsto} \qquad \text{circle}$$

We can now state that three points, $x_{1\cdots3}$ on this line with the representations $X_{1\cdots3}$ must invert to give three points, $X'_{1\cdots3}$ on this circle. Let the general point on the line be represented by X; we know, from the above incidence relations, that

$$X \wedge X_1 \wedge X_2 \wedge X_3 = 0$$

and thus, if X' is a general point on the circle, we know that

$$X' \wedge X'_1 \wedge X'_2 \wedge X'_3 = 0$$

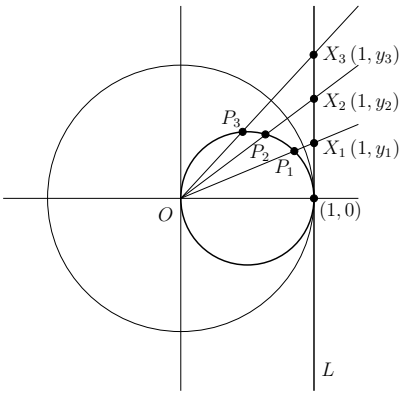


Figure 3.1: An illustration of the inversion of points on the line $x = 1$ (L) in the unit circle centred on the origin. It produces a circle centred on $(\frac{1}{2}, 0)$ and with radius $\frac{1}{2}$. The points at infinity on the line L map to the origin.

We can see this by performing an inversion upon the line via a reflection in e

$$e(X \wedge X_1 \wedge X_2 \wedge X_3)e = eXe \wedge eX_1e \wedge eX_2e \wedge eX_3e$$

This gives a very useful form for the equation of a circle. We derived it for a special case but since we know that we can dilate and translate as we wish, it must in fact be true for a completely general circle. Thus if $X_{1\cdots3}$ are *any* three points, the equation of the circle passing through these points is

$$X \wedge C = X \wedge X_1 \wedge X_2 \wedge X_3 = 0 \tag{3.5}$$

Where we identify the trivector C as the representation of the circle itself. C is formed by wedging any three points on the circle together. We will see later that all such trivectors are equivalent up to scale.

Treating this as a general incidence relation for a circle and by sandwiching it between e it is clear that we will, in general, obtain another circle under inversion since each $X_i' = eX_ie$ will be another general point in the plane.

This only fails if X_1', X_2', X_3' happen to be collinear and, further, co-linearity will only occur if the original circle passes through the origin (as in the case we started with here).

We can see this if we consider what happens to the representation of the origin, $\bar{n}$ under inversion. If we sandwich $\bar{n}$ in between e we obtain $e\bar{n}e = n$ and hence we justify our association of n with the point at infinity (the usual result of inverting the origin). Therefore the incidence relation for a line may always be written in the form

$$X \wedge n \wedge X_1 \wedge X_2 = 0 \tag{3.6}$$

where X_1 and X_2 are the representations of any two finite points on the line. This explains the extra point we appeared to need in describing a line earlier. We can now see that

$$X \wedge X_1 \wedge X_2 \wedge X_3 = 0$$

actually describes a *circle* and, therefore, genuinely requires 3 points whilst a line is just a special case of a circle which passes through the point at infinity. To summarise, any line, L, is represented by a trivector of the form

$$L = X_1 \wedge X_2 \wedge n \tag{3.7}$$

and similarly a circle, C, is represented by a trivector of the form:

$$C = X_1 \wedge X_2 \wedge X_3 \tag{3.8}$$

where no X_i is a multiple of n. This close relationship between circles and lines and the interpretation of a line as a circle passing through the point

at infinity is not new. The whole field of *inversive geometry* [7] in the plane has this as its basis. In the above example when we took a line to a circle through the origin we were effectively inverting in the unit circle centred on the origin, as illustrated in figure 3.1. The entire of inversive geometry in the plane can be encapsulated by the operation of reflecting points (X) and lines (L) in general circles (C), i.e.

$$\begin{aligned} X' &= CXC \\ L' &= CLC \end{aligned}$$

The huge advantage that we have in our conformal framework is that precisely the same equations can be used in 3d. So much so that, considering that 3d spheres are merely circles in 4d space, we can state that a sphere is represented as

$$S = X_1 \wedge X_2 \wedge X_3 \wedge X_4 \tag{3.9}$$

where X_1, X_2, X_3, X_4 are the null-vector representations of points on its surface. Conventionally, much of inversive geometry is described by complex numbers and so the jump from the plane to higher-dimension spaces is rarely made.

3.5 Vectors and 2-blades

We have seen that we may use null vectors in our 5d space to represent points in Euclidean 3-space. In particular we identify n and $\bar{n}$ with the point at infinity and the origin respectively. In our 5d space there clearly exist vectors

which do not square to zero. The interpretation of such vectors will be discussed later.

The term *blade* in GA is used to refer to quantities which can be written as the wedge product of vectors. For example a r-blade can always be written as $A_1 \wedge A_2 \wedge \cdots \wedge A_r$. It is important to distinguish this from an r-vector which may be any linear combination of r-blades. This is important since, in 5d, not all bivectors can be written as 2-blades; for example $e_1 e_2 + e_3 e_4$ cannot be written in the form $A \wedge B$ with A and B being vectors.

We start by postulating that $A \wedge B$ represents the pair of points with null-vector representations A and B. This is clear when we consider the incidence relation

$$X \wedge A \wedge B = 0$$

which is only true in general for $X = A$ or $X = B$. If we accept that $A \wedge B$ represents two points then we should develop an algorithm to extract the individual null-vectors A and B.

3.5.1 Extracting A and B from $A \wedge B$

In this section we explain how to extract A and B from $A \wedge B$ using a method of *projectors*. Throughout we shall assume that A and B have been normalised so that $A \cdot n = -1$. We start by considering the 2-blade $T = A \wedge B$ and form

$$F = \frac{1}{\beta} A \wedge B \tag{3.10}$$

where $\beta > 0$ and $\beta^2 = T^2$, so that $F^2 = 1$ if $\beta^2 \neq 0$. We now use F to define two *projector* operators

$$\begin{aligned} P &= \frac{1}{2}(1+F) \\ \tilde{P} &= \frac{1}{2}(1-F) \end{aligned} \tag{3.11}$$

where $\tilde{P}$ denotes the normal reversion operation applied to P. Note that $P^2 = P$, which can be be verified

$$\begin{aligned} PP &= \frac{1}{4}(1+F)(1+F) \\ &= \frac{1}{4}(1+2F+1) = \frac{1}{2}(1+F) \end{aligned} \tag{3.12}$$

Similarly, we can show that $\tilde{P}\tilde{P} = \tilde{P}$. These properties justify calling these operators *projectors*, borrowing the term from physics. An equally important property is that $P\tilde{P} = \tilde{P}P = 0$ which, again, is easy to prove

$$P\tilde{P} = \frac{1}{4}(1+F)(1-F) = \frac{1}{4}(1-1) = 0 \tag{3.13}$$

and similarly for $\tilde{P}P$. We may now see what effect these projectors have on A and B

$$\begin{aligned} PA &= \frac{1}{2}\left[1 + \frac{1}{\beta}A \wedge B\right] A \\ &= \frac{1}{2}\left[A + \frac{1}{\beta}(A \wedge B)A\right] \\ &= \frac{1}{2}\left[A + \frac{1}{\beta}(A \cdot B)A\right] \\ &= \frac{1}{2}(A - A) = 0 \end{aligned} \tag{3.14}$$

since $(A \wedge B)A = (A \wedge B) \cdot A = -A^2 B + (A \cdot B)A = (A \cdot B)A$ ($A^2 = 0$) and $A \cdot B = -\beta$. This follows from $\beta^2 = (A \wedge B) \cdot (A \wedge B) = -A^2 B^2 + (A \cdot B)^2$ and the facts that $A^2 = B^2 = 0$ and $A \cdot B$ must be negative as seen in equation 2.8.

Using similar working we can also show the results of P and $\tilde{P}$ acting on A and B

$$PA = 0 \tag{3.15}$$

$$PB = B \tag{3.16}$$

$$\tilde{P}A = A \tag{3.17}$$

$$\tilde{P}B = 0 \tag{3.18}$$

The next step is to consider the vector obtained by dotting $A \wedge B$ with n.

$$(A \wedge B) \cdot n = -n \cdot (A \wedge B) = -(n \cdot A)B + (n \cdot B)A = (B - A) \tag{3.19}$$

using the fact that A and B are normalised points such that $A \cdot n = B \cdot n = -1$. It therefore follows that we have

$$P[(A \wedge B) \cdot n] = P(B - A) = B \tag{3.20}$$

$$-\tilde{P}[(A \wedge B) \cdot n] = -\tilde{P}(B - A) = A \tag{3.21}$$

We note also that since $AP = \tilde{P}A = A$ it follows that $\tilde{P}AP = \tilde{P}\tilde{P}A = \tilde{P}A$. Similar relations hold for BP etc., so that we have

$$\tilde{P}AP = \tilde{P}A$$

$$PA\tilde{P} = 0$$

$$PB\tilde{P} = PB$$

$$\tilde{P}BP = 0$$

which means that we can also write the projections as two-sided operations.

Thus from a 2-blade $A \wedge B$ we can extract the two points A and B that it represents via

$$\begin{aligned} A &= -\tilde{P}\left[(A \wedge B) \cdot n\right] \equiv -\tilde{P}\left[(A \wedge B) \cdot n\right] P \\ B &= P\left[(A \wedge B) \cdot n\right] \equiv P\left[(A \wedge B) \cdot n\right] \tilde{P} \end{aligned} \tag{3.22}$$

We will see later that when we perform intersection operations that yield two points the two points in question can then be found using the formulæ in equation 3.22. Usefully we do not have to solve a quadratic equation as we would do using conventional approaches.

3.6 Trivectors

We have already seen that there are two classes of object represented by trivectors. If P, Q, R are null vectors in our 5d space representing points in 3d space then trivectors of the form

$$C = P \wedge Q \wedge R$$

represent circles. Recall that it is specifically a circle passing through points represented by P, Q and R. Trivectors of the form

$$L = P \wedge Q \wedge n$$

represent lines, specifically that line passing through the points represented by P and Q.

In GA it is often found that the operation of taking the dual, that is multiplication by the pseudoscalar, is useful and often has physical or geometric

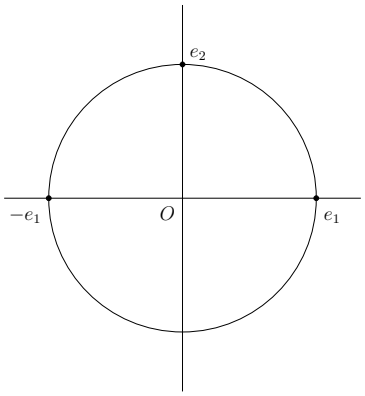


Figure 3.2: Unit circle with three key points marked

significance. The dual of an element is always just another element of the algebra and does not live in a separate 'dual' or 'tangent' space. Below we shall consider the dual operation with respect to the representations of circles and lines.

3.6.1 Circles as trivectors

Here we will show that taking the dual of the trivector representing a circle gives rise to a useful alternative representation which naturally encodes both the centre and radius. Let us first of all work in the plane so that our conformal space is 4-dimensional, and is $\mathcal{A}(3,1)$, having basis vectors e_1, e_2, e and $\bar{e}$.

We start with the unit circle in the plane and take as three points on it those shown in figure 3.2. For any unit length vector, $\hat{x}$, we know that $F(\hat{x}) = \frac{1}{2}(n + 2\hat{x} - \bar{n}) = (\hat{x} + \bar{e})$. In particular we have

$$F(e_1) \wedge F(e_2) \wedge F(-e_1) = 2e_1 e_2 \bar{e}$$

and hence the trivector $C = 2e_1e_2\bar{e}$ represents the unit circle. In the plane the pseudoscalar, which we shall write as I_4, is given by $I_4 = e_1e_2e\bar{e}$ and so the dual of C, which we write as C^*, can be shown to be given by

$$C^* = CI_4 = 2e = (n + \bar{n}) \tag{3.23}$$

We know that $X \wedge C = 0$ is the incidence relation for the circle and that $X \wedge C = 0$ can be rewritten as

$$X \cdot (CI_4) = 0 \quad \Leftrightarrow \quad X \cdot C^* = 0$$

Note that C^* is the dual of a trivector in a 4d space and is, therefore, a vector. This suggests a very useful alternative representation for a circle, or a sphere when generalised to higher dimensions.

We know from equation 2.8 that for any two normalised point representations A and B

$$A \cdot B = -\frac{1}{2}(a-b)^2$$

and thus, if X represents a point on a circle and B represents its centre, we know that we can write

$$X \cdot B = -\frac{1}{2}(x-b)^2 \equiv -\frac{1}{2}\rho^2$$

where ρ is the radius of the circle. For a normalised point representation X this implies that

$$X \cdot (B - \frac{1}{2}\rho^2 n) = 0$$

since $X \cdot n = -1$. Comparing this with $X \cdot C^*$ we see that provided we normalise C^* after taking the dual (so that $C^* \cdot n = -1$), then we find

$$C^* = B - \frac{1}{2}\rho^2 n \tag{3.24}$$

The vector C^*, therefore, encodes in a neat fashion the centre and radius of the circle in the plane.

Also note that if we take our $(n+\bar{n})$ as the representation of the dual of the unit circle centred on the origin and rotate, translate and dilate the expression, we get precisely the same result as in equation 3.24. We can see this by first taking our unit circle at the origin, $C_o^* = n+\bar{n}$ and rotating it. We have seen that rotation rotors leave n and $\bar{n}$ invariant, so C_o^* remains unchanged. Now dilate this with a dilation rotor $D_\alpha = e^{\frac{\alpha}{2}e\bar{e}}$ where $\rho = e^{-\alpha}$ is the dilation factor. We have seen previously that $D_\alpha n \tilde{D}_\alpha = e^{-\alpha}n$ and $D_\alpha \bar{n} \tilde{D}_\alpha = e^{\alpha}\bar{n}$, so that

$$C_o^* \rightarrow W = e^{-\alpha}n + e^{\alpha}\bar{n} \tag{3.25}$$

Now, translate by a using a translation rotor, $T_\alpha = 1 + \frac{na}{2}$. Again, using previous results, we know that n is left invariant and $-\frac{1}{2}\bar{n}$ is taken to $F(a)$, giving us

$$W \rightarrow Z = e^{-\alpha}n - 2e^{\alpha}F(a) \tag{3.26}$$

Normalising this to give us our new C^*, where $C^* \cdot n = -1$, means dividing Z above by $-2e^{\alpha}$ (since $A = F(a)$ satisfies $A \cdot n = -1$) leaving us with

$$C^* = -\frac{1}{2}e^{-2\alpha}n + A \tag{3.27}$$

and we can now see that this is precisely equation 3.24 with A representing the centre and $\rho = e^{-\alpha}$ as the radius.

So far we have talked about circles in the plane. We shall now look at the treatment of circles at general positions and orientations in space. Firstly we

note that since we can think of C as the wedge product of the representations of three points on the circle, it follows that the *plane* which the circle lies in is $C \wedge n$.

In deriving equation 3.24 for the circle in the plane we used the pseudoscalar for the 4d space. It therefore seems plausible that when we move to general circles in 3d the rôle of this pseudoscalar will be taken by the plane in which the circle lies.

We firstly define the *unit* plane I_c by

$$I_c = \frac{n \wedge C}{\sqrt{(-[n \wedge C]^2)}} \tag{3.28}$$

since $(n \wedge C)$ always squares to give a negative scalar. In future we will find that it is convenient to always take the line, plane, circle, sphere etc. of unit magnitude. The dual of the circle C is then given by the analogous equation to equation 3.24 where we assume that given C, which can be 'unit', we form C^* and then normalise such that $C^* \cdot n = -1$. That is

$$C^* \equiv CI_c = B - \frac{1}{2}\rho^2 n \tag{3.29}$$

where B is again the centre of the circle, ρ is the radius. Note that the 'dual' in this case is with respect the 'unit' plane in which the circle lies. The proof of this is in most respects identical to the previous proof for the plane but where before we used I_4, we now use the unit plane. This plane is proportional to $e_1 \wedge e_2 \wedge \bar{n} \wedge n$, and is indeed identical to I_4. We can then rotate, dilate and translate as before and note that $R(C_o I_c)\tilde{R} = RC_o\tilde{R}RI_c\tilde{R}$ – and hence our new dual – has the form given in equation 3.29. The dual is formed by taking

the product of the transformed circle with the transformed plane it lies in. We also note that we can find the radius of this general circle very simply by squaring C^*

$$\begin{aligned}(C^*)^2 &= (B-\frac{1}{2}\rho^2 n)^2 \\ &= -\rho^2 B\cdot n = \rho^2 \end{aligned} \tag{3.30}$$

using the facts that $B^2 = 0$, $n^2 = 0$ and $B\cdot n = -1$. From this it then follows that $B = C^* + \frac{1}{2}(C^*)^2 n$. To summarise, from the vector form of any general circle, we can easily obtain the centre and radius as follows

$$(C^*)^2 = \rho^2 \tag{3.31}$$

$$B = C^*\left[1+\frac{1}{2}C^* n\right] \tag{3.32}$$

Note here that the above relations assume C^* is normalised such that $C^*\cdot n = -1$ since $C^*\cdot n = B\cdot n = -1$, as we assume B is a normalised null vector.

While the above formulation is indeed useful, we will now see that there is a far more elegant way of finding the centre of a circle in 3d. The centre of a circle, C, is also given by simply reflecting the point at infinity, n, in the circle, i.e.

$$CnC \tag{3.33}$$

To prove that CnC gives the centre we can return to our circle with its centre at the origin and a unit radius. We saw earlier in this section that we can write this circle as $C = 2e_1 e_2 \bar{e}$ and, again, we can define a *unit circle*, $\hat{C}$, as

$\hat{C} = e_1 e_2 \bar{e}$. Thus we see that

$$\begin{aligned} \hat{C}n\hat{C} &= e_1 e_2 \bar{e} n e_1 e_2 \bar{e} \\ &= -e_2 \bar{e} n e_2 \bar{e} \\ &= -\bar{e} n \bar{e} \\ &= \bar{e}(e+\bar{e})\bar{e} = e - \bar{e} \\ &= \bar{n} \end{aligned} \tag{3.34}$$

which is indeed the origin and therefore the centre of the circle. Having proved that the result holds for this simple case, we can now rotate, dilate and translate our circle to give any other circle and the result will still hold. Suppose we apply a rotor, R, which is a composition of rotors which rotate, dilate and translate, to take our circle, C, to any other general circle, $C' = RC\tilde{R}$. Then we see that

$$C'nC' \propto RC\tilde{R}(RnR)RC\tilde{R} \propto R(CnC)\tilde{R} \tag{3.35}$$

since we have seen previously that $RnR \propto n$ for R composed of rotations, translations and dilations. Equation 3.35 thus tells us that the rotated origin, $R(CnC)\tilde{R}$, is indeed given by $C'nC'$. This type of simple proof, i.e. proving a result for some simple case at the origin and generalising via rotations, translations and dilations, is a nice feature of the conformal framework. There will be many other examples of the XaX formulation producing something interesting in subsequent sections.

In many physical systems we find that GA is very useful in that it exposes how *observables* usually arise by sandwiching some value between a multi-

vector and its inverse or reverse. For example, in quantum mechanics, we get the spin current in 3d from a Pauli spinor, ϕ, via $s = \phi e_3 \tilde{\phi}$. We see then that this example of sandwiching n between a circle and its reverse (the reverse of a circle is itself up to a scale factor) is an example of this principle at work in the conformal setting.

3.6.2 Lines as trivectors

Next we consider the circles passing through the point at infinity, which we have already seen are lines. Again we begin by asking ourselves what the dual of a line, L, actually encodes. We suspect the answer to this question will provide relevant information about the line. As with circles, let us begin by considering lines in the plane. As a starting point we take the line L parallel to the x-axis and distance d from the x-axis. This line is given by wedging together any two points on the line, say $A = F(de_2)$, $B = F(e_1 + de_2)$, and n;

$$\begin{aligned} L &= F(de_2) \wedge F(e_1 + de_2) \wedge n \\ &= \frac{1}{4}(d^2 n + 2de_2 - \bar{n}) \wedge \left(\sqrt{(1+d^2)}n + 2(e_1 + de_2) - \bar{n}\right) \wedge n \\ &= -\frac{1}{2} e_1 \wedge n \wedge \bar{n} - de_1 \wedge e_2 \wedge n \end{aligned} \tag{3.36}$$

If we then take the dual of L, again with respect to the pseudoscalar, I_4, for our 2d conformal space, we see that

$$\begin{aligned} L^* &= LI_4 = -(\frac{1}{2} e_1 E + de_1 e_2 n) e_1 e_2 e\bar{e} \\ &= e_2 + dn = e_1 I_2 + dn \end{aligned} \tag{3.37}$$

using the identities, $e_1E = Ee_1$, $e_2E = Ee_2$, $ne\bar{e} = n$ and $Ee\bar{e} = -2$. Thus, we see that the dual of the line in the plane gives the dual, with respect to the pseudoscalar in Euclidean space, $I_2 = e_1e_2$, of the Euclidean unit vector in the direction of the line plus dn where d is the perpendicular distance of the line from the origin. This holds for any d, and if we apply rotors to rotate our line (noting again that $RnR = n$) we see that we can extend our proof to any line in the plane, therefore giving

$$L^* = \hat{m}I_2 + dn \tag{3.38}$$

where $\hat{m}$ is the unit vector in the direction of the line and d is the perpendicular distance of the line from the origin.

Recalling that $n \cdot \bar{n} = 2$, we can extract d easily from L^* as follows

$$d = \frac{1}{2}L^* \cdot \bar{n} \tag{3.39}$$

Similarly we can then extract $\hat{m}$ as

$$\hat{m} = -[L^* - \frac{1}{2}(L^* \cdot \bar{n})n]I_2 \tag{3.40}$$

As with circles, it should be straightforward to extend this work on lines from the plane into space. However, unlike the circle case, we do not have an obviously specified plane with which to define a 4d pseudoscalar, so instead we look at the dual of the line with respect to the pseudoscalar for the whole 5d space. Clearly we multiply a 3-vector with a 5-vector and get a 2-vector. Once again, consider the same line that we considered above. We look at the

dual of this line with $I \equiv I_5$

$$\begin{aligned} L^* &= -(\frac{1}{2}e_1 \wedge n \wedge \bar{n})I - d(e_1 e_2 n)I \\ &= e_2 e_3 - d(e_1 \wedge e_2)In \\ &= e_1 I_3 + [((de_2) \wedge e_1))I]n \end{aligned} \tag{3.41}$$

since $In = nI$. Clearly the first term gives us the unit vector in the direction of the line and the second term gives us its moment. As this holds for any d, if we rotate our line we can produce any line in space and we see that the above can therefore be generalised to hold for any such line. More specifically we write the dual of the line L as

$$L^* = \hat{m}I_3 + [(a \wedge \hat{m})I_3]n$$

where $\hat{m}$ denotes the unit vector (in 3d) in the direction of the line and a is any 3d point on the line. This is analogous to writing the line in terms of Plücker coordinates, where 3 of the coordinates give the line's direction and the other 3 give its moment about the origin.

3.7 4-Vectors

We have already seen that 4-vectors in the conformal setting can represent both spheres and planes. Having seen how we interpret the duals for circles and lines, it will now be easy to extend these arguments to spheres and planes.

3.7.1 Spheres as 4-vectors

Given any 4 points whose 5d representations are P, Q, R, S, the sphere through those points is given by the 4-vector Σ

$$\Sigma = P \wedge Q \wedge R \wedge S$$

We know that $X \wedge \Sigma = 0$ for any X lying on the sphere. This can be rewritten as

$$X \cdot (\Sigma I) = 0 \quad \Longrightarrow \quad X \cdot \Sigma^* = 0$$

where $\Sigma^* = \Sigma I$ is the dual to Σ and, hence, is a vector. We can now follow precisely the same workings given for the circle to show that the dual representation of the sphere naturally encodes the centre and the radius.

If X is a point on a sphere and C is its centre we know that we can write

$$X \cdot C = -\frac{1}{2}(x-c)^2 \equiv -\frac{1}{2}\rho^2$$

where ρ is the radius of the sphere. For a normalised point X ($X \cdot n = -1$) this therefore implies that

$$X \cdot (C - \frac{1}{2}\rho^2 n) = 0$$

Comparing this with $X \cdot \Sigma^*$ we see that provided we normalise Σ^* after taking the dual (such that $\Sigma^* \cdot n = -1$), then we will find

$$\Sigma^* = C - \frac{1}{2}\rho^2 n \tag{3.42}$$

As before, the dual vector Σ^* encodes the centre and radius of the sphere. One may use Σ or Σ^* depending on whether it is most useful to specify the sphere

by 4 points lying on it or by its centre and radius. Given a Σ^* (via taking the normalised form of the dual of $\Sigma = P \wedge Q \wedge R \wedge S$) we can immediately get the radius and centre in the manner outlined earlier for the circle

$$(\Sigma^*)^2 = \rho^2 \tag{3.43}$$

$$C = \Sigma^* \left[1 + \frac{1}{2}\Sigma^* n\right] \tag{3.44}$$

As we saw with the case of circles, there is also a more elegant means of extracting the centre of a sphere given Σ or $S = \Sigma^*$. The method is to reflect the point at infinity, n, in the sphere so that the centre, C, is given by

$$C = SnS = \Sigma n \Sigma \tag{3.45}$$

To show this we consider the sphere of radius 1 centred on the origin, $\Sigma_o = F(e_1) \wedge F(e_2) \wedge F(e_3) \wedge F(-e_1)$. We can expand this to give $2e_1e_2e_3\bar{e}$, so that the dual, S, is given by the particularly concise expression $2e$; we therefore take the 'unit' sphere to be e. We then see that

$$SnS = ene = \bar{n} \tag{3.46}$$

which is indeed the origin, the centre of the sphere in this case. Now if we rotate, dilate, and translate our sphere at the origin to any other sphere in space, say S', then, reflecting n in our new sphere gives

$$\begin{aligned} S'nS' &= (RS\tilde{R})n(RS\tilde{R}) = (RS\tilde{R})Rn\tilde{R}(RS\tilde{R}) \\ &= R[SnS]\tilde{R}. \end{aligned} \tag{3.47}$$

Since $R[SnS]\tilde{R}$ is the transformed origin, i.e. the new centre of the sphere, we see that the formula $S'nS'$ does indeed give the centre.

3.7.2 Planes as 4-vectors

A plane, Φ, passing through the 3 points whose 5d representations are P, Q, R, is given by

$$\Phi = P \wedge Q \wedge R \wedge n$$

The physical quantities that we might want to extract from such a 4-vector are clearly the normal to the plane and the perpendicular distance of the plane from the origin. We now investigate how the dual form of the plane helps us to do this. Once again we start by considering the plane $z = d$ which is parallel to the x-y plane and distance d from it. We can represent Φ by

$$\begin{aligned}
\Phi &= F(de_3) \wedge F(e_1 + de_3) \wedge F(e_2 + de_3) \wedge n \\
&= \frac{1}{8}\{(2de_3 - \bar{n}) \wedge (2[e_1 + de_3] - \bar{n}) \wedge (2[e_2 + de_3] - \bar{n}) \wedge n\} \\
&= de_1 \wedge e_2 \wedge e_3 \wedge n - \frac{1}{2} e_1 \wedge e_2 \wedge \bar{n} \wedge n \\
&= de_1 \wedge e_2 \wedge e_3 \wedge n - e_1 \wedge e_2 \wedge e \wedge \bar{e}
\end{aligned} \tag{3.48}$$

where the second line in equation 3.48 follows because wedging with n removes any term containing n. Then it is simple to show that the dual of Φ is given by

$$\Phi^* = \Phi I = dn + e_3 \tag{3.49}$$

This holds for any d. If we rotate this plane via a rotor we know that the proof must hold, therefore since $Rn\tilde{R} = n$ and $Re_3\tilde{R} = \hat{n}$, where $\hat{n}$ is the new rotated normal to the plane, the general equation for the dual of the plane is given by

$$\Phi^* = dn + \hat{n} \tag{3.50}$$

Note that the above assumes that we have normalised the dual such that $[\Phi^*]^2 = 1$; we can ensure this by normalising the plane such that $\Phi^2 = 1$. Therefore, given 3 points on the plane, P, Q, R, we form the normalised plane Φ and its dual Φ^*, and we can then extract $\hat{n}$ and d as follows

$$d = \frac{1}{2}\Phi^* \cdot \bar{n} \tag{3.51}$$

$$\hat{n} = \Phi^* - \frac{1}{2}(\Phi^* \cdot \bar{n})n \tag{3.52}$$

Whether we use the 4-vector form or the *normal-distance* form of the plane will depend upon the problem we are solving.

3.8 Intersections

In this section we outline the various ways of intersecting objects within the conformal model. In general we shall use an operator termed the *meet*[29], represended by the symbol $\vee$, which given two objects A and B gives their intersection.

3.8.1 Intersecting spheres with spheres or planes

Let us consider the intersection of two spheres Σ_1 and Σ_2 where they may intersect in a circle, at a point or not at all. Suppose we take the formula for the meet[34] (where now we use $*$ to indicate multiplication by the pseudoscalar I_n):

$$C = \Sigma_1 \vee \Sigma_2 = \left[\langle \Sigma_1 \Sigma_2 \rangle_{2n-r-s}\right]^* \tag{3.53}$$

$2n-r-s=10-4-4=2$, so that the dual quantity will have grade $5-2=3$ — generally, this will give the trivector representing the circle of intersection. We can tell whether we have a circle, a point intersection or no intersection according to whether

$$C^2>0 \quad C^2=0 \quad \text{or} \quad C^2<0 \tag{3.54}$$

In the case of $C^2>0$ we can extract the centre and radius according to section 3.7.1. If we also attempt to extract the centre and radius via these same formulæ from C where $C^2=0$, we will find that the circle will have zero radius and its centre will be the point of tangency of the two spheres. Similarly, attempting to extract the radius and centre from C in the case $C^2<0$ (i.e. no intersection) leads to an imaginary radius and a centre which lies on the shortest line joining the surfaces of the spheres (i.e. that joining the centres). If the two spheres have the same radii, it is the midway point on this line.

The above all follows through if instead of having a second sphere, Σ_2, we have a plane, Φ_2 – we again get a trivector for our intersection object via the meet and the sign of the square of this trivector tells us whether the two objects are tangent, intersect in a circle or do not intersect at all.

3.8.2 Intersecting spheres with circles or lines

Let us now intersect a sphere Σ_1 (4-blade) with a circle C_2 (3-blade). According to our meet formulæ our

intersection is a 2-blade, B, given by

$$B=\Sigma_1\vee C_2=\left[\langle\Sigma_1 C_2\rangle_{2n-r-s}\right]^* \tag{3.55}$$

where $2n-r-s = 10-4-3 = 3$, so that the dual object has grade 2. We have already seen that these 2-blades represent 2 points — precisely as we would expect, since an intersecting sphere and circle will do so at 2 points. Now we again look at the sign of the resulting 2-blade, B, and we will find that there are zero, one or two points of intersection when

$$B^2 = 0 \quad B^2 < 0 \quad \text{or} \quad B^2 > 0.$$

respectively. We have seen earlier that given a bivector B, such that $B^2 > 0$, of the above form, we can extract the two points of intersection via the projectors given in equations 3.22. If $B^2 = 0$ we cannot form the projector, but it is trivial to find the representation of the point of contact, X, in this case using the following

$$X = BnB$$

i.e. for a 2-blade of the form $W = P \wedge Q$, reflecting n in W, WnW, would give us the midpoint of the line joining P and Q — for our case where $B^2 = 0$, the construction BnB will therefore give us a representation of the point of intersection. These results can easily be shown by considering simple cases at the origin and then extending the proof via rotors as previously.

Precisely the same working holds if we replace our circles above with lines — the meet again gives a 2-vector whose square tells us whether there are 2, 1 or no intersections, and from which the intersection points can be obtained easily. We will return to the intersections of lines with spheres when we later consider reflections of lines in spheres.

3.8.3 Intersecting planes with planes, circles and lines

Consider two planes Φ_1 and Φ_2; taking the meet gives

$$L = \Phi_1 \vee \Phi_2 = \left[\langle \Phi_1 \Phi_2 \rangle_{2n-r-s}\right]^* \tag{3.56}$$

where $2n-r-s = 10-4-4 = 2$, so that the dual object has grade 3 — as we would expect, if the planes intersect to give a line. We are able to tell whether the planes intersect by looking at the sign of L^2 — if $L^2 = 0$ we know that the planes are parallel and do not intersect, if $L^2 > 0$, the planes intersect in the line L.

Now consider a plane Φ_1 and a circle C_2; we take the meet of these two objects to give

$$B = \Phi_1 \vee C_2 = \left[\langle \Phi_1 C_2 \rangle_{2n-r-s}\right]^* \tag{3.57}$$

where $2n-r-s = 10-4-3 = 3$, so that the dual object has grade 2 — the plane and the circle intersect at a maximum of two points, and the 2-blade, B, encodes these two points as with the sphere-circle intersection. Once again we can assert that there are 2, 1 or 0 intersections according to whether $B^2 > 0, B^2 = 0, B^2 < 0$. In the case of two intersections, the points are extracted from B by projectors as before; in the case of tangency, the one point of contact is obtained by taking BnB.

It is worth thinking about what happens when the circle C_2 lies in the plane Φ_1 so that the intersection is C_2 itself. As one might expect, in this case there *is no* grade 3 part of $\Phi_1 C_2$. In the case of the $z = 0$ plane and the unit circle lying in the plane and centred on the origin this can easily be

confirmed:

$$\Phi_1 = F(e_1) \wedge F(e_2) \wedge F(-e1) \wedge n \propto e_1 e_2 e \bar{e}$$

$$C_2 = F(e_1) \wedge F(e_2) \wedge F(-e1) \propto e_1 e_2 \bar{e}$$

thus

$$\Phi_1 C_2 \propto e_1 e_2 e \bar{e} e_1 e_2 \bar{e} = e \text{ hence } \langle \Phi_1 C_2 \rangle_3 = 0$$

We can also note that, in this case, the dual of $\Phi_1 C_2$ with respect to $e_1 e_2 e \bar{e}$ is indeed C_2.

If we now replace our circle by a line, L_2, it is clear that the meet will still give us a 2-blade, B however we know that the line and the plane intersect in at most one location, so should we not be looking for a vector rather than a 2-blade? The answer is that if the plane and the line intersect, and the meet gives us B, then B is always of the form

$$B = X \wedge n$$

where X is the representation of the point of intersection. This can be proved easily by again considering a simple case at the origin. If $B^2 > 0$ the line and plane intersect in a point, if $B^2 = 0$ the line and plane do not intersect and if $B = 0$ the line lies in the plane. If there is one point of intersection so that B is of the above form, we can extract the three-dimensional point of intersection, $x = x^i e_i$, $i = 1, 2, 3$ (and hence X), by simply equating x^i to the coefficient of the $e_i \wedge n$ term or by using the following expansion

$$x = (B \wedge \bar{n}) \cdot E \tag{3.58}$$

where $E = n \wedge \bar{n}$ as given earlier.

3.8.4 Intersecting circles with circles and lines

Consider two circles, C_1 and C_2; taking their meet gives

$$X = C_1 \vee C_2 = \left[\langle C_1 C_2 \rangle_{2n-r-s}\right]^* \tag{3.59}$$

where $2n-r-s = 10-3-3 = 4$, so that the dual object has grade 1. We know, however, that the intersection of two circles is at most 2 points (only possible if they lie in the same plane), so how do we get two points from our grade 1 object? In fact we find that the following is true

$$\begin{aligned} C_1 \vee C_2 &= X \text{ where } X^2 = 0 \text{ if circles have one intersection} \\ C_1 \vee C_2 &= X \text{ where } X^2 \neq 0 \text{ if circles have no intersection} \\ C_1 \vee C_2 &= 0 \quad \text{if circles have two intersections.} \end{aligned} \tag{3.60}$$

In the case where the meet gives zero and we know there are two intersections, these can easily be found by intersecting the plane of one of the circles with the other circle, i.e.

$$B = C_1 \vee (C_2 \wedge n) = \left[\langle C_1 (C_2 \wedge n) \rangle_{2n-r-s}\right]^* \tag{3.61}$$

where $2n-r-s = 10-3-4 = 3$, so that the dual object has grade 2. The two points of intersection can be extracted from the 2-blade B using equation 3.22.

If we now replace C_2 by a line L_2 we see that we again get a grade 1 object

when we take the meet, and the situation above is exactly replicated, i.e.

$$\begin{aligned} C_1 \vee L_2 &= X \text{ where } X^2 = 0 \text{ if circle and line have one intersection} \\ C_1 \vee L_2 &= X \text{ where } X^2 \neq 0 \text{ if circle and line have no intersection} \\ C_1 \vee L_2 &= 0 \quad \text{if circle and line have two intersections} \end{aligned} \tag{3.62}$$

As before, in the case where the meet gives zero the two intersections can easily be found by intersecting the plane of one of the circles with the line. It is also interesting to note here that in the case where the circle and the line do not intersect, with the meet giving a vector, X, which is not null, the sign of X^2 tells us whether the line passes through the circle ($X^2 < 0$) or does not pass through the circle ($X^2 > 0$) – such simple checks can often be useful in graphics applications.

Given that we have had a little difficulty with circles intersecting circles, we might expect some slight difficulties with lines and lines. It turns out that many interesting constructions emerge when we start to consider the intersections between two lines; these will be discussed in the following section.

3.8.5 Intersecting lines with lines

Let us consider two lines, L_1 and L_2. Taking the meet of these two lines gives

$$X = L_1 \vee L_2 = \left[\langle L_1 L_2 \rangle_{2n-r-s}\right]^* \tag{3.63}$$

where $2n - r - s = 10 - 3 - 3 = 4$, so that the dual object has grade 1. We might expect that if the lines intersect at a point, the meet, X, will give this

intersection point – however, this is not the case. We find that the following is true

$$\begin{aligned} L_1 \vee L_2 &= 0 \quad \text{if the lines intersect} \\ L_1 \vee L_2 &\propto n \quad \text{if the lines do not intersect} \end{aligned} \tag{3.64}$$

We therefore have a simple of way for checking for intersecting lines, but if the meet gives us zero so that we know there is one intersection point, we can no longer find this point in a fully covariant way by intersecting one line with the a plane defined by the other line, since such a plane is not uniquely defined. We could, in practise, intersect one line with the plane formed by the other line and the origin, but then if the other line passes through the origin, this will not work, leaving us a non-covariant procedure and one which entails us forming conditionals for a number of cases. We would instead like to look for a method which works covariantly – such a method exists, and in the process of describing it, we see a number of other useful constructions.

Take our arbitrary lines, L_1 and L_2 (assume they are normalised, such that $L_1^2 = L_2^2 = 1$). Suppose we reflect line L_1 in line L_2 — this statement is not well-defined in a conventional sense, but in GA we have seen that reflection of an object in another object is indeed well defined and is brought about by sandwiching the object to be reflected between the object that it is being reflected in. So our reflected line L_1' is given by

$$L_1' = L_2 L_1 L_2$$

as shown in figure 3.3. The operation of reflection is grade preserving if we

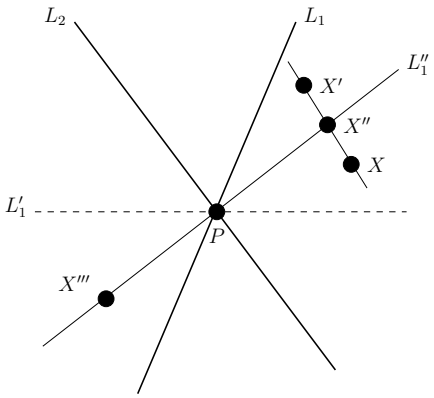


Figure 3.3: The rotation of intersecting lines to produce two lines intersecting at right angles, via the construction $L_1 - L_2 L_1 L_2$.

are dealing with blades, and therefore we know that that results in another line. This construction will work for any two lines, but let us now suppose that our lines intersect at a point; we can then expect that the reflected line L_1' will be the line formed by the intersection point, represented by P, and any point on L_1 reflected in L_2. This is indeed what L_1' is. Now, however, it becomes possible to form the line which is perpendicular to L_2, passing through the intersection point, P, and in the plane defined by L_1 and L_2 via the following

$$L_1'' = L_1 - L_2 L_1 L_2 \tag{3.65}$$

Clearly what we are doing here is rotating one line in the plane defined by the two lines to become perpendicular to the other line. We will return to this rotor description later. Once we have two perpendicular lines which intersect, we can find the point of intersection relatively easily. Take any arbitrary point representation X and reflect it in L_1'' (assuming again that we

have normalised L_1'') via $X' = L_1'' X L_1''$, then take the midpoint of X and X'. We know that this must lie on the line L_1''

$$X' = L_1'' X L_1'' \quad X'' = \frac{1}{2}(X + X')$$

It is straightforward to verify explicitly that X'' is the representation of the true mid-point of the points represented by X and X' plus some multiple of n. Now we reflect X'' in L_2 to give X''' and again take the midpoint — the midpoint must now give us the intersection point P plus some multiple of n;

$$X''' = L_2 X'' L_2 \quad P' = \frac{1}{2}(X'' + X''')$$

we then extract the null vector corresponding to the representation of our intersection point P via

$$P = \frac{-(P' n P')}{2(P' \cdot n)^2}$$

which has the effect of removing the multiple of n. This construction is independent of X and is a beautiful illustration of the ability to manipulate objects using geometric algebra. Although it appears involved, it can be implemented very easily ($\bar{n}$ can be used as the point X) and is an entirely covariant way of intersecting two lines. Of course, in practice, one can also check to see if there is at least one line that does not pass through the origin ($\bar{n} \wedge L = 0$ if L passes through the origin); if there is one, for example L_1, we can then form the plane $\bar{n} \wedge L_1$ and intersect this with L_2. If both lines pass through the origin then the intersection point is the origin. Note that we can use precisely the same type of argument to extract the plane formed by two intersecting lines. For example, take any arbitrary point X, reflect it in the

line L_1 via L_1XL_1 so that the midpoint of this line is given by P, where P is given (up to some additional multiple of n) by

$$P = \frac{1}{2}(X + L_1XL_1)$$

P must clearly lie on L_1, thus the plane formed by the two lines must be given by

$$L_2 \wedge P = L_2 \wedge \frac{1}{2}(X + L_1XL_1) \tag{3.66}$$

Again, $\bar{n}$ can be used for our point X in real computations. The derivations given here are entirely covariant and the same constructions will intersect 'lines' in different geometries; we will illustrate this towards the end of the thesis by considering three-dimensional hyperbolic geometry.

Recall that previously we found the reflection of L_1 in L_2 via $L_2L_1L_2$. Now note that we can rewrite this as

$$L_2L_1L_2 = (L_2L_1)L_1\widetilde{(L_2L_1)} = RL_1\tilde{R} \tag{3.67}$$

since $\widetilde{(L_2L_1)} = \tilde{L}_1\tilde{L}_2 = L_1L_2$. Thus we see that the quantity L_2L_1 acts as a rotor (if the lines are normalised) which rotates through twice the angle between the lines about an axis through the intersection point and perpendicular to the plane containing the lines. In fact, when we write this reflection as a rotation, there is nothing to insist that our lines must intersect. If L_1 and L_2 do not intersect, then $L_2L_1L_2$ will still perform a *reflection* of L_1 in L_2, but we are now able to interpret exactly what this reflection means for non-intersecting lines if we regard the operator as a rotor. It turns out that the rotor L_2L_1 is the product of a rotation rotor and a translation rotor – the rotation is in the

plane normal to the common perpendicular of the lines and the translation is along the common perpendicular, so that one line is taken onto and then through the other. We can show that it is possible to write the product L_2L_1 as

$$L_2L_1 = (\cos\theta + \hat{B}\sin\theta)(1 + dn) \tag{3.68}$$

where d is the three-dimensional vector representing the length and direction of the common perpendicular, $\hat{B} = \hat{d}I_3$ (with $\hat{d} = d/\sqrt{(d^2)}$) and θ is the angle between the lines as measured when translated to lie in the same plane along d. We see that the above is a combination of a rotation rotor and a translation rotor; we will see later that rotors which take one object (of the same grade) into another object are often formed in the way we have outlined here for lines.

3.9 Chapter summary

In this chapter we have examined how the Conformal Model can be used to represent geometric primitives such as lines, planes, spheres, circles and point pairs. We have also shown how, within the Conformal Model, rotors can be applied to these objects just as they would be applied to points. The Conformal Model outlined in this chapter will form a foundation upon which much of the results of this thesis will be based.

"As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realised that a large part of my life from then on was going to be spent in finding mistakes in my own programs."

— Maurice Wilkes

4

LibCGA — A Library for Implementing GA-based Algorithms

Any GA-based algorithm must have a method of implementation to be truly useful in any engineering sense. In order to provide a solid base for the development of such algorithms it was decided that a good general-purpose GA library should be created in order to allow for rapid prototyping of GA-based solutions. The library was called `libcga`.

4.1 Requirements

There were a number of design aims when designing the `libcga` software library.

- Fast — The library should be numerically efficient.
- User-friendly — The library Application Programmer's Interface (API) should be convenient and simple to use.
- Compact — The library should be small enough to be embedded within

a larger solution.

- Thin — The library should be sufficiently low-level so as to introduce little penalty for wrapping it in a higher-level API.

These design aims were, by their nature, highly coupled. Research into CGA would influence the design of the library and design of the library would promote and influence the direction of the research. Because of this I decided to make use of the spiral model of software development where the software is incrementally improved from an initial prototype to adapt to changing design parameters and new features.

It was decided that the implementation should follow the Object-Oriented Programming (OOP) methodology due to the natural mapping between multivectors and operators in GA and objects and object-operators in OOP. The C language was chosen because of the availability of high-quality optimising compilers and the relative closeness of the language to machine code, minimising the amount of intermediary code output from the compiler.

One way in the C language of writing object-orientated programs is to make use of *opaque data structures*. An internal structure type is defined and all library API functions communicate with the library passing a pointer to the structure as the first argument. All access to the structure is done by the library through this pointer so the library user need not know of the structure's layout. This is an example of *data hiding*, a common feature of object-orientated programming.

$$A^G = \begin{bmatrix}
+A_0 & +A_1 & +A_2 & +A_3 & -A_{12} & -A_{13} & -A_{23} & -A_{123} \\
+A_1 & +A_0 & +A_{12} & +A_{13} & -A_2 & -A_3 & -A_{123} & -A_{23} \\
+A_2 & -A_{12} & +A_0 & +A_{23} & +A_1 & +A_{123} & -A_3 & +A_{13} \\
+A_3 & -A_{13} & -A_{23} & +A_0 & -A_{123} & +A_1 & +A_2 & -A_{12} \\
+A_{12} & -A_2 & +A_1 & +A_{123} & +A_0 & +A_{23} & -A_{13} & +A_3 \\
+A_{13} & -A_3 & -A_{123} & +A_1 & -A_{23} & +A_0 & +A_{12} & -A_2 \\
+A_{23} & +A_{123} & -A_3 & +A_2 & +A_{13} & -A_{12} & +A_0 & +A_1 \\
+A_{123} & +A_{23} & -A_{13} & +A_{12} & +A_3 & -A_2 & +A_1 & +A_0
\end{bmatrix}$$

Figure 4.1: Example product matrix for the geometric product in $\mathcal{A}(3,0)$. $A_{ij\ldots k}$ is the element of A proportional to $e_i e_j \ldots e_k$.

4.2 Overview

Any multivector M in $\mathcal{A}(p,q)$ can be formed from a linear combination of all possible basis vector products up to grade $p+q$. For $\mathcal{A}(4,1)$, the highest grade object is $e_1 e_2 e_3 e\bar{e}$ so

$$M = a_1 + a_2 e_1 + a_3 e_2 + \ldots + a_7 e_1 e_2 + \ldots + a_{32} e_1 e_2 e_3 e\bar{e}$$

Initially `libcga` stored a multivector as the vector $[a_1, a_2, \ldots a_{32}]'$ implemented as an array. Later optimisations added the ability to keep track of which grades were present in the multivector and this will be addressed later.

The various multivector products may be found by expanding out the terms of the product and simplifying using a product table similar to Table

2.1. Alternatively, one may view the multivectors A, B, C as the vector representations $\mathbf{A}, \mathbf{B}, \mathbf{C}$ and calculate $C = AB$ using

$$\mathbf{C} = A^G \mathbf{B}$$

where A^G is a 32×32 matrix whose elements depend on the operator used (in this case the geometric product) and the elements of $\mathbf{A}$. A little thought makes it clear that all linear operators can be expressed in this form and so all the operators we have defined can thus be expressed. An example matrix for the geometric product in $\mathcal{A}(3,0)$ generated by Gaigen is shown in figure 4.1.

This method is somewhat sub-optimal since $n \times n$ matrix-vector multiplications require $O(n^2)$ operations. Techniques were developed to reduce this general form to a set of more compact, efficient operations and these will be outlined in section 4.3.3.

4.3 Implementation Details

4.3.1 Coding style

In common with many modern APIs it was decided the library should posses an object-oriented API. One way in the C language of writing object-orientated programs is to make use of *opaque data structures*. An internal structure type is defined and all library API functions communicate with the library passing a pointer to the structure as the first argument. All access to the structure is done by the library through this pointer so the library user need not know

```
1  struct {
2    int num_widgets;
3    char widget_name[255];
4  } widget_s;
5  typedef struct widget_s widget_t;
6
7  widget_t* widget_new() {
8    return (widget_t*)malloc(sizeof(struct widget_s));
9  }
10 void widget_delete(widget_t *self) {
11   free(self);
12 }
13 void widget_set_num(widget_t *self, int num) {
14   self->num_widgets = num;
15 }
```

Figure 4.2: Object-orientation in C.

of the structure's layout. This is an example of *data hiding*, a common feature of object-orientated programming.

An example of how to create constructor, destructor and member access functions in C is given in Figure 4.2. From the programmers' point of view they only have to use the API function passing the handle returned by widget_new() as the first argument.

4.3.2 Product Table Generation

Product tables were generated with a Perl script in which n-vector components were represented by a string of digits. For example, the trivector $e_1e_4e_5$ was represented as '`145`'. Similarly $e_1e_5e_4$ was reordered and represented as '`-145`' so that the possible representations conformed to a set of basis elements.

Any geometric product between two blades was then calculated using the algorithm represented in figure 4.3. The algorithm can be outlined as followed:

1. Sort first component numerically by exchanging neighbours and alternate sign once for each swap.
2. Reverse sort second component numerically by exchanging neighbours and alternate sign once for each swap.
3. If the last digit of the first component and first digit of the second component match, change sign as appropriate to the square of the components and remove.
4. If there are more pairs to match, goto step 3.
5. Concatenate components and output.

Other product tables were computed from the geometric product. For example $a \cdot b = 0.5(ab + ba)$.

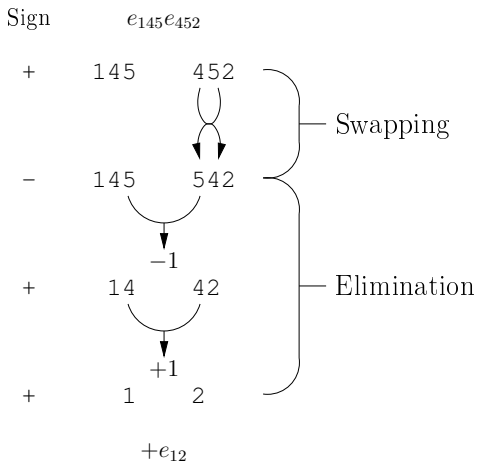


Figure 4.3: Example of finding that $e_{145}e_{452} = e_{12}$ with $e_5^2 = -1$, $e_4^2 = 1$.

These product tables were then used to generate optimal *n*-vector, *m*-vector product routines in C which operated directly on the multivector components.

4.3.3 Grade Tracking

Table 4.1 shows the number of floating-point multiplications to compute the geometric product of a pure *r*-vector and pure *s*-vector. As you can see even for a worst case bivector-bivector product only 100 multiplications are required compared to 1024 for the general product. This suggests that significant speedups can be obtained if only those grades present in the multivector are considered.

The script used to generate the product tables above could also be used to generate specific *n*-vector, *m*-vector product tables. These specific tables often used far fewer flops than required for the general operator. This effectively

		s					
		0	1	2	3	4	5
r	0	1	5	10	10	5	1
	1	5	25	50	50	25	5
	2	10	50	100	100	50	10
	3	10	50	100	100	50	10
	4	5	25	50	50	25	5
	5	1	5	10	10	5	1

Table 4.1: Multiplication count for finding the geometric product of an r-vector and s-vector.

exploited the sparseness of the product matrix for single-grade products.

When looking for an optimisation strategy the following properties are desirable:

- Transparent to the programmer — the core `libcga` API.
- Straightforward to implement.
- Generic (i.e. not limited to $\mathcal{A}(4,1)$.
- Provide significant reduction in floating point operation count.

It was clear that optimised product implementations provided significant speedups but required the programmer to know in advance which grades were present in a multivector (not always possible if the multivector is ultimately due to user input). The solution was to represent the general multi-

vector M as a sum of single-grade objects

$$M = \langle M \rangle_0 + \langle M \rangle_1 + \ldots + \langle M \rangle_n$$

where $\langle M \rangle_i$ represents taking the grade i component of M and thus the product of the multivectors A and B is

$$\begin{aligned} AB = & \langle A \rangle_0 \langle B \rangle_0 + \langle A \rangle_0 \langle B \rangle_1 + \ldots \\ & + \langle A \rangle_1 \langle B \rangle_0 + \langle A \rangle_1 \langle B \rangle_1 + \ldots \\ & + \langle A \rangle_n \langle B \rangle_{n-1} + \langle A \rangle_n \langle B \rangle_n \end{aligned}$$

Now let G_A be the set of grades present in A and G_B be the set of grades present in B so that

$$\begin{aligned} \langle A \rangle_i &= 0 \quad \text{if } i \notin G_A \\ \langle B \rangle_i &= 0 \quad \text{if } i \notin G_B \end{aligned}$$

hence it can be said that

$$\langle A \rangle_i \langle B \rangle_j = 0 \quad \text{if } i \notin G_A \text{ or } j \notin G_B$$

and need not be computed. If G_A and G_B are sufficiently small with respect to $\{0 \ldots 5\}$ then significant advantage may be obtained.

In order to implement this it was necessary to record in the multivector which grades were present. The most time and space efficient method to do this in C is via a bit-mask. This technique keeps a 32-bit unsigned integer called the *grade mask*. If bit n in the mask is set then grade n is present. This does limit the maximum grade to 32-bits but this can be extended to 64-grade

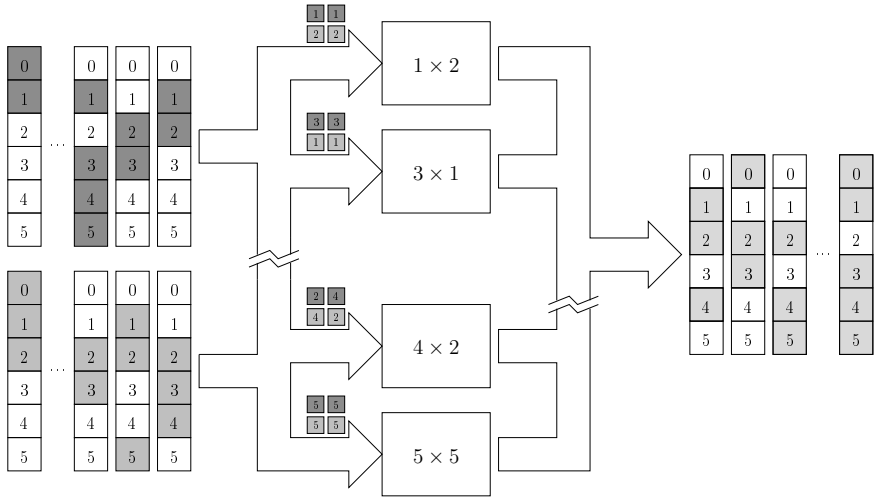


Figure 4.4: The method of grade tracking represented graphically. The shaded numbers represent the grades present in each multivector.

by using a 64-bit integer on certain systems (e.g. the IA-64 next generation Intel processor) or even higher order grades by utilising multi-word masks.

The advantage of using the object-orientated programming methodology now became clear, in that a grade mask field could be added to the opaque `multiv_t` data type without changing the API. Also, since the program using `libcga` only referenced this structure through a pointer, existing programs were *binary compatible*, that is they didn't have to be recompiled.

The product function itself was modified to check for the presence of each grade-pair in the two input multivectors and only call the appropriate set of single-grade routines. It was the job of each single-grade product function to set the appropriate grade mask bits in the output multivector. This approach is shown diagrammatically in figure 4.4

4.4 Visualising Objects within the Algebra

Many CGA visualisation solutions exist[50]. The `libcgadraw` companion library to `libcga` was created as a convenience library for visualising the output of algorithms implemented with `libcga`. The library uses an approach similar to that of the CLUDraw library whereby the results of a GA-based algorithm may be displayed directly on the screen. The output of GA-based algorithms may be categorised into a number of geometric primitives they represent. Below is a selection of the visualisation algorithms used. In section 5.3 we will develop some further algorithms for visualising the output of algorithms working in non-Euclidean geometries.

4.4.1 Point Pairs

Point pairs are represented by the wedge product of the representations of the points. That is to say that the point pair (a,b) is represented by $F(a) \wedge F(b)$. Using the method of projectors in section 3.5.1 the representations of a and b can be extracted and plotted as shown in algorithm 4.5.

4.4.2 Lines

We render lines by intersecting them with some sphere centred on the origin and drawing a line between the two points of intersection. The radius of the sphere used effectively provides a upper limit to the distance a line may be from the origin while still being rendered. If the line is further from the origin than the radius of the sphere it will not intersect the sphere.

Require: T, a representation of a point-pair.
1: $T' := \frac{1}{\sqrt{T^2}}T$
2: $P := \frac{1}{2}(1 + T')$
3: $B := P[T \cdot n]$
4: $A := -\tilde{P}[T \cdot n]$
5: $a := F^{-1}(A)$
6: $b := F^{-1}(B)$
7: glBegin(GL_POINTS);
8: glVertex(a); glVertex(b);
9: glEnd();

Figure 4.5: Extracting and rendering a point pair.

This is in actual fact a desirable property since it allows for a 'horizon' to be set. Lines, being infinite in extent, have end points at infinity which cannot easily be expressed within OpenGL. Instead, by setting the radius of the sphere to be sufficiently large we may approximate such lines sufficiently for rendering in many problems. The specific method is given in algorithm 4.6.

4.4.3 Planes

A plane may easily be rendered via its dual representation outlined in section 3.7.2. Recall that, for a plane P, if $p = PI \equiv P^*$ then

$$p = \hat{n} + dn$$

where $\hat{n}$ is the plane normal and d is its perpendicular distance from the origin.

Require: L, a representation of a line.
Require: $r_{\max}$, maximum distance from origin to render lines.
1: $r := 1$
2: **repeat**
3: intersect := false
4: $S :=$ representation of sphere centred on origin radius r.
5: $T := L \vee S$
6: **if** $T \neq 0$ **then**
7: intersect := true
8: Extract point pair from T to a, b as in algorithm 4.5.
9: **end if**
10: $r := r + 1$
11: **until** intersect = true **or** $r > r_{\max}$
12: **if** intersect = true **then**
13: glBegin(GL_LINES);
14: glVertex(a); glVertex(b);
15: glEnd();
16: **end if**

Figure 4.6: Rendering the representation of a line, L.

4.4.4 Circles

Recall that the circle passing through the point-representations P_1, P_2 and P_3 is represented by $C = P_1 \wedge P_2 \wedge P_3$ and the plane common to these points is $P_1 \wedge P_2 \wedge P_3 \wedge n$ hence we can easily find the plane of the circle, P via

$$P = C \wedge n$$

To find the radius and centre of the circle we use the dual form of the circle given in 3.6.1 and [35]

$$C^* = B - \frac{1}{2}\rho^2 n$$

Require: C, a representation of a circle.
1: $\rho := \sqrt{(C^*)^2}$
2: $b := F^{-1}\left(C^*\left[1 + \frac{1}{2}C^* n\right]\right)$
3: $P := C \wedge n$
4: $\hat{n} := \text{spatial}(P^*)$
5: Draw circle in plane with normal $\hat{n}$ centre b, radius ρ.

Figure 4.7: Rendering the representation of a circle, C.

where $C^* \equiv CI_4$ here (the dual of C with respect to the pseudoscalar for the plane), B is the centre of the circle and ρ is the radius. The radius can be found as in section 3.6.1.

The final algorithm (for three dimensions) is outlined in algorithm 4.7. In this algorithm the function spatial() extracts the components of the argument which contain no part of e or $\bar{e}$.

4.4.5 Spheres

Finally spheres may be rendered simply by noting that circles are two-dimensional spheres and all the formulæ for obtaining centres and radii hold true. Extracting the null-vector representation of the centre, $B = F(b)$, and the radius, ρ, of a particular sphere Σ is therefore straightforward:

$$B = \Sigma n \Sigma$$

and

$$\rho^2 = (\Sigma^*)^2 .$$

4.5 Chapter summary

In this chapter we outlined the design decisions for the GA library used to generate results in this thesis. Many of the figures in this thesis were generated by code based upon this library. Further we have examined techniques for extracting useful properties of objects within the conformal model with a view to visualising them on a display device.

"Equations are just the boring part of mathematics. I attempt to see things in terms of geometry."

— Stephen Hawking

5

Non-Euclidean Techniques

In this chapter we will extend the conformal model to cover non-Euclidean geometries. There has been some work in extending the conformal model to spherical geometry[38] and here we present a method for extending it to hyperbolic geometry. We also briefly outline how such an approach may be modified to encompass the work on spherical geometry.

Euclidean geometry, the geometry of the plane, was first defined[19] by *Euclid's Postulates*:

1. A straight line segment can be drawn joining any two points.
2. Any straight line segment can be extended indefinitely in a straight line.
3. Given any straight line segment, a circle can be drawn having the segment as radius and one endpoint as centre.
4. All right angles are congruent.

5. If two lines are drawn which intersect a third in such a way that the sum of the inner angles on one side is less than two right angles, then the two lines inevitably must intersect each other on that side if extended far enough.

This last postulate is equivalent to what is known as the *parallel postulate*, which states loosely that two parallel lines will never meet, even at infinity. In this chapter we shall explore hyperbolic geometry, one of the simplest geometries that satisfy all but this last postulate.

5.1 Hyperbolic Geometry

Hyperbolic geometry is usually represented in two dimensions on the *Poincaré disc*. This is a unit disc centred on the origin onto which all space is mapped. A metric is defined within the disc[7], along with a set of congruence transformations. The boundary circle of the disc represents the points at infinity. Everywhere outside the disc is inaccessible to the hyperbolic geometry. Straight lines, called *d*-lines[7], are represented by circular arcs which erupt normal to the boundary circle. Perhaps one of the most famous examples of this geometry is given in Escher's *Circle Limit* series of wood prints, an example of which is recreated in figure 5.1. In these prints infinite tessellations of hyperbolic space are represented mapped to the Poincaré disc and they show the circular nature of *d*-lines, although Escher was unaware of the Poincaré disc model at the time the prints were made.

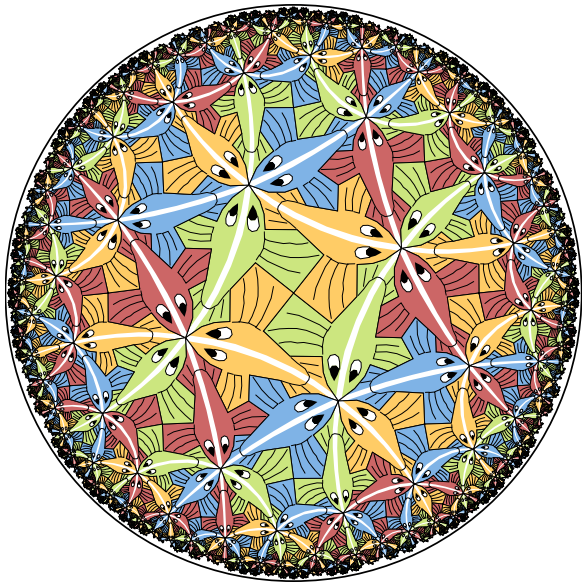


Figure 5.1: A re-creation of Escher's *Circle Limit III*, a depiction of hyperbolic geometry on the Poincaré disc. Taken from [18].

5.2 Extending the Conformal Model

In this section we shall extend the conformal model we have developed to represent Euclidean geometry to the hyperbolic geometry of the Poincaré disc and show how many results which are tedious to prove using existing metric-based derivations become simple and, in some cases, obvious using the GA-based approach.

All rigid-body transformation (i.e. rotation and translation) rotors in the Euclidean approach described in previous chapters leave n invariant, i.e. $RnR\tilde{} = n$ for all such rotors R. They also leave $\bar{n}$ invariant. We have already identified n and $\bar{n}$ with the points at infinity and the origin respectively. We shall now show that, since a geometry may be defined by its congruence

transformations, changing which vectors are left invariant by rotors will produce a different geometry.

Suppose we choose to restrict the rotors such that they keep e invariant. Without loss of generality, we deal with a conformal extension of $\mathbb{R}^2$ and write down a set of four basis vectors

$$E_1 = e_1 \quad E_2 = e_2 \quad E_3 = e \quad E_4 = \bar{e} \tag{5.1}$$

and thus form the rotors $R_{k\ell} = \exp\left(\frac{\alpha}{2} E_k \wedge E_\ell\right)$ with $k, \ell \in \{1, 2, 3, 4\}$. Applying them to e via $R_{k\ell} e \tilde{R}_{k\ell} \equiv R_{k\ell} e R_{\ell k}$, we find the bivector generators of the rotors which preserve e are $\bar{e}e_1$, $\bar{e}e_2$ and $e_1 e_2$. The latter just corresponds to rotations in the $e_1 e_2$ plane, and hence the former two must be the generators of translations. We propose, therefore, that a rotor which translates the origin to the vector x must be given by a multivector of the form

$$T_x = \exp\left(\frac{f(r)}{2} \bar{e}\hat{r}\right) \tag{5.2}$$

where $r = |x|$, $\hat{r} = x/|x|$ and $f(r)$ is some function of r yet to be determined. Noting that $(\bar{e}e_1)^2 = (\bar{e}e_2)^2 = +1$ and therefore $(\bar{e}\hat{r})^2 = +1$ we can take the power series expansion of T_x and collect like-coefficients to obtain

$$T_x = \cosh\left(\frac{f(r)}{2}\right) + \bar{e}\hat{r} \sinh\left(\frac{f(r)}{2}\right) \tag{5.3}$$

We now consider how to represent the origin. Our choice is restricted in that the origin must differ from the point at infinity and, to retain isotropy, must not contain components parallel to e_1 or e_2. Either n or $\bar{n}$ would be a suitable choice to investigate; we choose a multiple of $\bar{n}$ to retain compatibility with the Euclidean case.

As with the Euclidean case we wish to impose a normalisation condition on the null-vectors, $X = F_e(x)$, such that

$$X \cdot e = -1 \tag{5.4}$$

where we use $F_e(x)$ to represent the mapping defined by the geometry generated by the rotors which preserve e. We would also prefer if we could contain some relation to our mapping for Euclidean geometry. Hence we continue by specifying that the origin is represented by null-vectors parallel to $-\bar{n}$.

We can now find the representation of the general point x as the translation along x of the origin. Writing $c = \cosh\left(\frac{f(r)}{2}\right)$ and $s = \sinh\left(\frac{f(r)}{2}\right)$

$$\begin{aligned} F_e(x) &= T_x\,(-\bar{n})\,\tilde{T}_x & (5.5)\\ &= [c + \bar{e}\hat{r}s]\,(-\bar{n})\,[c - \bar{e}\hat{r}s] & (5.6)\\ &= -c^2\bar{n} + 2sc\hat{r} + s^2 n. & (5.7) \end{aligned}$$

Letting $C = \cosh(f(r))$ and $S = \sinh(f(r))$ gives $c^2 = (C+1)/2$, $s^2 = (C-1)/2$, $sc = S/2$ and hence

$$\begin{aligned} F_e(x) &= \frac{1}{2}n(C-1) - \frac{1}{2}\bar{n}(C+1) + S\hat{r}\\ &= \frac{1}{2}\left[(C-1)n + 2S\hat{r} - (C+1)\bar{n}\right] & (5.8) \end{aligned}$$

As required, $(F_e(x))^2 = 0$ and $F_e(x) \cdot e = -1$.

It remains to choose a sensible form for $f(r)$. We seek to choose $f(r)$ such that the representation $F_e(x)$ is similar to our Euclidean representation $F(x)$ since this will allow us to use many of the same techniques we developed for the Euclidean case. We can rewrite our Euclidean representation in terms of

r and $\hat{r}$ as

$$F(x) = \frac{1}{2\lambda^2}(r^2 n + 2\lambda r\hat{r} - \lambda^2 \bar{n}) \tag{5.9}$$

where λ has been introduced as a fundamental unit of length to make the equation dimensionally consistent. We shall discuss this later.

If we wish that $F_e(x)$ be similar to $F(x)$ then we have the conditions

$$\frac{S}{C+1} = \frac{\sinh(f(r))}{\cosh(f(r))+1} = \frac{r}{\lambda} \tag{5.10}$$

and

$$\frac{C-1}{S} = \frac{\cosh(f(r))-1}{\sinh(f(r))} = \frac{r}{\lambda} \tag{5.11}$$

so the mapping function becomes

$$F_e(x) = \frac{C+1}{2\lambda^2}\,[x^2 n + 2\lambda x - \lambda^2 \bar{n}] \tag{5.12}$$

$$= \frac{\cosh(f(r))+1}{2\lambda^2}\,[x^2 n + 2\lambda x - \lambda^2 \bar{n}]o \tag{5.13}$$

which has a degree of similarity to the expression for $F(x)$. Further, assuming r and λ are positive, we can see from equation 5.10 that $r < \lambda$ since $\sinh(A) < 1 + \cosh(A)$ for all A.

Given equations 5.10 and 5.11, we can eliminate $\sinh(f(r))$ to give

$$\frac{\cosh(f(r))-1}{\cosh(f(r))+1} = \frac{r^2}{\lambda^2} \tag{5.14}$$

and therefore $\cosh(f(r)) = (\lambda^2 + r^2)/(\lambda^2 - r^2)$. Substituting into either 5.10 or 5.11 gives

$$f(r) = \sinh^{-1}\left(\frac{2\lambda r}{\lambda^2 - r^2}\right) \tag{5.15}$$

and hence we can form the following expressions for $\sinh(f(r))$ and $\cosh(f(r))$

$$\sinh(f(r)) = \frac{2\lambda r}{\lambda^2 - r^2} \quad \text{and} \quad \cosh(f(r)) = \frac{2\lambda^2}{\lambda^2 - r^2} - 1. \tag{5.16}$$

Inserting these into equation 5.13 gives the final form of the non-Euclidean mapping function

$$F_e(x) = \frac{1}{\lambda^2 - x^2}(x^2 n + 2\lambda x - \lambda^2 \bar{n}) \tag{5.17}$$

We can also show, by substituting the results in equation 5.16, that the form of the translation rotor given in equation 5.3 can also be written as

$$T_x = \frac{1}{\sqrt{\lambda^2 - x^2}}(\lambda + \bar{e}x) \tag{5.18}$$

Some discussion of the relevance of λ is worthwhile here. Notice that in order for the translator to remain real-valued, $x^2 \leq \lambda^2$. We can never therefore translate the origin outside of a circle radius λ centred upon it. The value of λ gives the distance to the boundary of a region of inaccessible space from the origin. It is in effect a circular boundary to the geometry. This circle corresponds directly to the unit-circle boundary in the Poincaré disc representation if $\lambda = 1$ and to simple dilations of the Poincaré representation if $\lambda \neq 1$. To maintain compatibility with the Poincaré representation, we usually set λ to be unity.

The rotor in equation 5.18 is, by construction, the rotor which takes us from the origin to position x. Interestingly translations in hyperbolic geometries do not commute; that is, moving a position vector a along another vector b does not, in general, give the same result as moving b along a. This can be

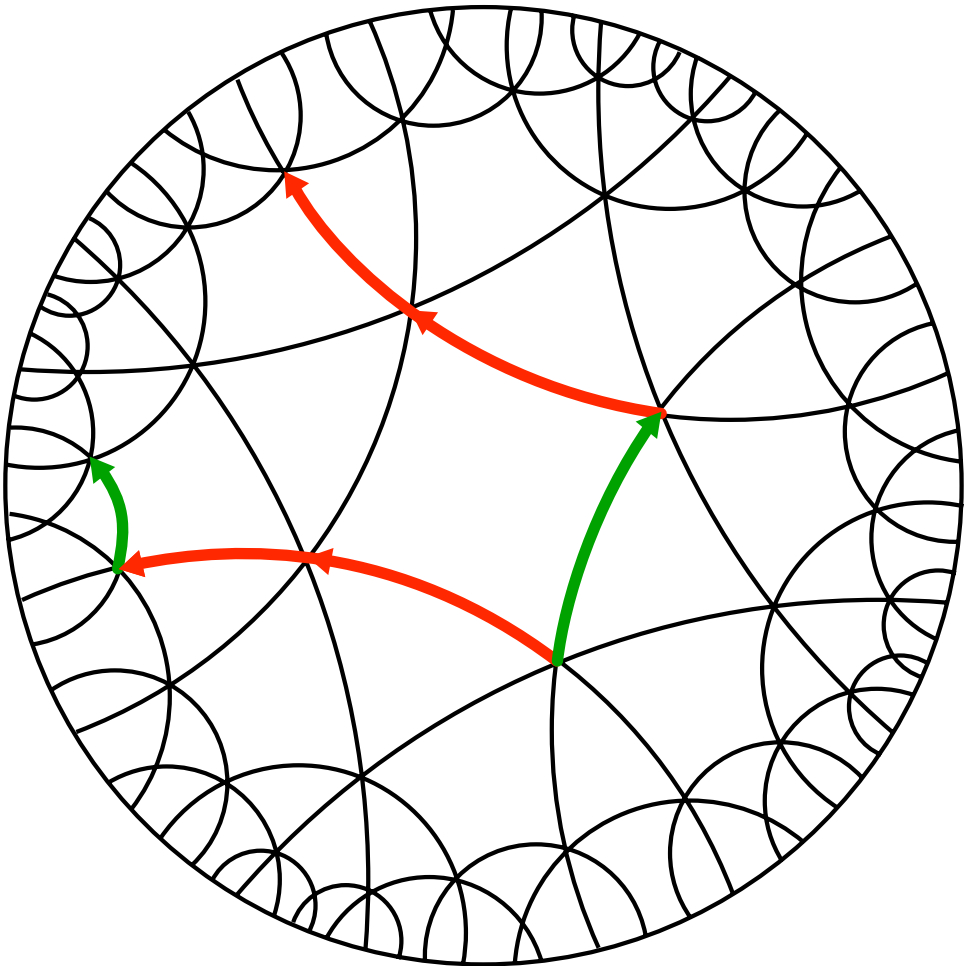


Figure 5.2: An illustration of how translation, interpreted as movement along geodesics, in hyperbolic geometry is non-commutative.

seen by tracing the grid in figure 5.1 as shown in figure 5.2. In this figure we translate along the vector drawn in red twice and translate along the vector marked in green once. Depending on the order of application we arrive at different locations. It is therefore geometrically clear why, unlike the Euclidean case, the hyperbolic translation rotors T_x and T_y do not commute for two different positions x and y. It is also clear algebraically since

$$T_x T_y = \alpha(\lambda + \lambda\bar{e}(x+y) - \bar{e}xy\bar{e})$$

$$T_y T_x = \alpha(\lambda + \lambda\bar{e}(x+y) - \bar{e}yx\bar{e}).$$

These are only identical if $xy = yx$, i.e. if x and y are parallel. This means that T_{y-x} is *not* the rotor taking us from position x to position y. However, this is not a problem since we can always achieve this motion via going back through the origin, and forming

$$T_{x\mapsto y} = T_y T_{-x} \tag{5.19}$$

Since composition of rotors always produces another rotor, this means that we have the same freedom as in the Euclidean case to prove a relation we are interested in, at some special position and orientation, and then use the covariant rotor structure to generalise the result to general positions and orientations. Spatial rotations about the origin are of course achieved as before with rotors of the form $R = \exp\left(\frac{\theta}{2} e_1 e_2\right)$.

The final motion we should consider is the analogue of inversion. Unlike in the Euclidean case, inversion using reflection in e is now a fully covariant operation. Specifically, if R represents any combination of rotation and

translation, and A is some object in the space, we have

$$eAe \mapsto Re\tilde{R}RA\tilde{R}Re\tilde{R} = eRA\tilde{R}e = R(eAe)\tilde{R} \tag{5.20}$$

The last equality means that transforming A first and then reflecting is the same as reflecting and then transforming, which is what is required of a covariant operation. The availability of this reflection operation is very useful. The translation rotors discussed above clearly only allow us to move around within the interior of the disc $r < \lambda$. By reflection in e, as we shall see below, we are able to jump into a 'dual world' outside the disc.

Having achieved a representation function and discussed the set of motions, we should examine this new space in relation to the Poincaré disc, which we have claimed it is equivalent to with a view to justifying this claim. To do this we shall examine the distance function, i.e. how we assign a non-Euclidean distance function between points in the space.

If we consider a simple rotation, $\exp\left(\frac{\theta}{2}\hat{B}\right)$, where $\hat{B}$ is some unit spatial bivector, then we are used to the idea that θ is the correct measure of distance (here angular) to describe the transformation. Thus if we consider again the translation rotor in equation 5.3, $T_x = \exp\left(\frac{f(r)}{2}\bar{e}\hat{r}\right)$, we would expect that the correct distance measure to associate with it would be $f(r)$. This would be a viable option, except that for points close to the origin of the disc ($r \ll \lambda$) it is desirable that ordinary Euclidean notions of distance to apply, at least approximately. It is clear that $\bar{e} = (1/2)(n - \bar{n})$ and the $\bar{n}$ part of this, when exponentiated and applied as in equation 5.8, has no effect to first order on

the origin point $-\bar{n}$, whereas the n part does, i.e.

$$\begin{aligned} T_x(-\bar{n})\tilde{T}_x &\approx \left(1+\frac{f(r)}{2}\frac{(n-\bar{n})\hat{r}}{2}\right)(-\bar{n})\left(1-\frac{f(r)}{2}\frac{(n-\bar{n})\hat{r}}{2}\right) \\ &= \left(1+\frac{f(r)}{2}\frac{n\hat{r}}{2}\right)(-\bar{n})\left(1-\frac{f(r)}{2}\frac{n\hat{r}}{2}\right). \end{aligned} \tag{5.21}$$

This means that, to first order near the origin, the T_x rotor approximates in its actions the Euclidean translation rotor corresponding to distance $f(r)/2$ rather than $f(r)$, i.e.

$$T_x \approx 1+\frac{f(r)}{2}\frac{n\hat{r}}{2} \tag{5.22}$$

For this reason, we take the non-Euclidean distance between a point and the origin to be given by $f(r)/2$ rather than $f(r)$. Calling this non-Euclidean distance function $d(r)$, and using equation 5.16 along with the identity

$$\sinh\left(\frac{z}{2}\right) = \left[\frac{\cosh z - 1}{2}\right]^{\frac{1}{2}},$$

gives

$$d(r) = \sinh^{-1}\left(\frac{r}{\sqrt{\lambda^2 - r^2}}\right) \tag{5.23}$$

We note this approximates to r/λ for $r \ll \lambda$, i.e. we recover the Euclidean distance measured in units of λ.

This function gives us the distance of a point from the origin, but what about the distance between two general points, neither of which is at the origin? One of the major advantages of the conformal approach to Euclidean geometry is that it gives us an inner product formula for computing the distance between any two points, and we would hope that the same would be possible here. This is indeed the case. Let X be the null vector corresponding to the point $x = r\hat{r}$ using our representation, equation (5.9). Then, since

$n \cdot \bar{n} = 2$, we can rewrite equation 5.23 as

$$d(r) = \sinh^{-1}\left(\sqrt{-\frac{1}{2}X \cdot (-\bar{n})}\right) \tag{5.24}$$

Note that $X \cdot (-\bar{n})$ is the inner product of X with the null vector representing the origin. Two points in any general positions can be wound back using a common translation rotor so that one of them ends up at the origin. For example, let T_y be the translation rotor that takes Y back to the origin, then

$$X' = T_y X \tilde{T}_y \qquad Y' = T_y Y \tilde{T}_y = -\bar{n}$$

so that $X' \cdot Y' = (T_y X \tilde{T}_y) \cdot T_y Y \tilde{T}_y = X \cdot Y = X' \cdot (-\bar{n})$. Since we know that a function of $-\frac{1}{2}X' \cdot (-\bar{n})$ gives the distance between the two points from equation (5.24), we can now write this distance in terms of $X \cdot Y$. Note that at no stage in the process has the inner product between the null vectors changed (the inner product is rotor invariant); thus we have succeeded in defining a distance between general points in terms of inner products. If the general points are x and y with representatives X and Y, the expression for the distance function is thus

$$d(x,y) = \sinh^{-1}\left(\sqrt{-\frac{1}{2}X \cdot Y}\right) \tag{5.25}$$

which is a satisfyingly simple relationship. As in the Euclidean case, it is a monotonic function of $X \cdot Y$. If we take the inner product of X and Y we obtain, using equation 5.9,

$$\begin{aligned} X \cdot Y &= \frac{1}{(\lambda^2 - x^2)(\lambda^2 - y^2)}\left[-2\lambda^2 x^2 - 2\lambda^2 y^2 + 4\lambda^2 xy\right] \\ &= \frac{-2\lambda^2}{(\lambda^2 - x^2)(\lambda^2 - y^2)}(x-y)^2 \end{aligned} \tag{5.26}$$

Written in terms of the points themselves, the distance between the points is therefore

$$d(x,y) = \sinh^{-1}\left(\lambda\sqrt{\frac{(x-y)^2}{(\lambda^2-x^2)(\lambda^2-y^2)}}\right). \tag{5.27}$$

Armed with this distance function, we can now investigate geodesics in the disc. These are the lines that are 'straightest' in the geometry defined by the distance function – more precisely, the arc length along them is extremal. We will not give the details, but show some numerically computed examples in figure 5.2. In calculating these we have taken $\lambda = 1$, so the disc is now the unit disc ordinarily considered in the Poicaré model. We find that each geodesic is an arc of a circle, and that each of them asymptotically approaches the bounding unit circle at right angles. At this point we can start making contact with the description of the classical approach to the Poincaré disc[7]. The geodesics there are called *d-lines*. They are not justified in terms of being geodesics, but simply defined as being arcs of circles which cut the boundary at right angles. However the distance function given in [7] in terms of distance to the origin is

$$d(x,0) = \tanh^{-1}(|x|) \tag{5.28}$$

and therefore agrees with our identification of $d(r) = f(r)/2$, using equation 5.16 and taking $\lambda = 1$.

This distance function allows us to define a geometrically analogous dilator about the origin. Suppose we dilate some vector x about the origin so that $x \mapsto x'$. We would expect that, given a dilation factor of a, $d(x',0) = a\,d(x,0)$

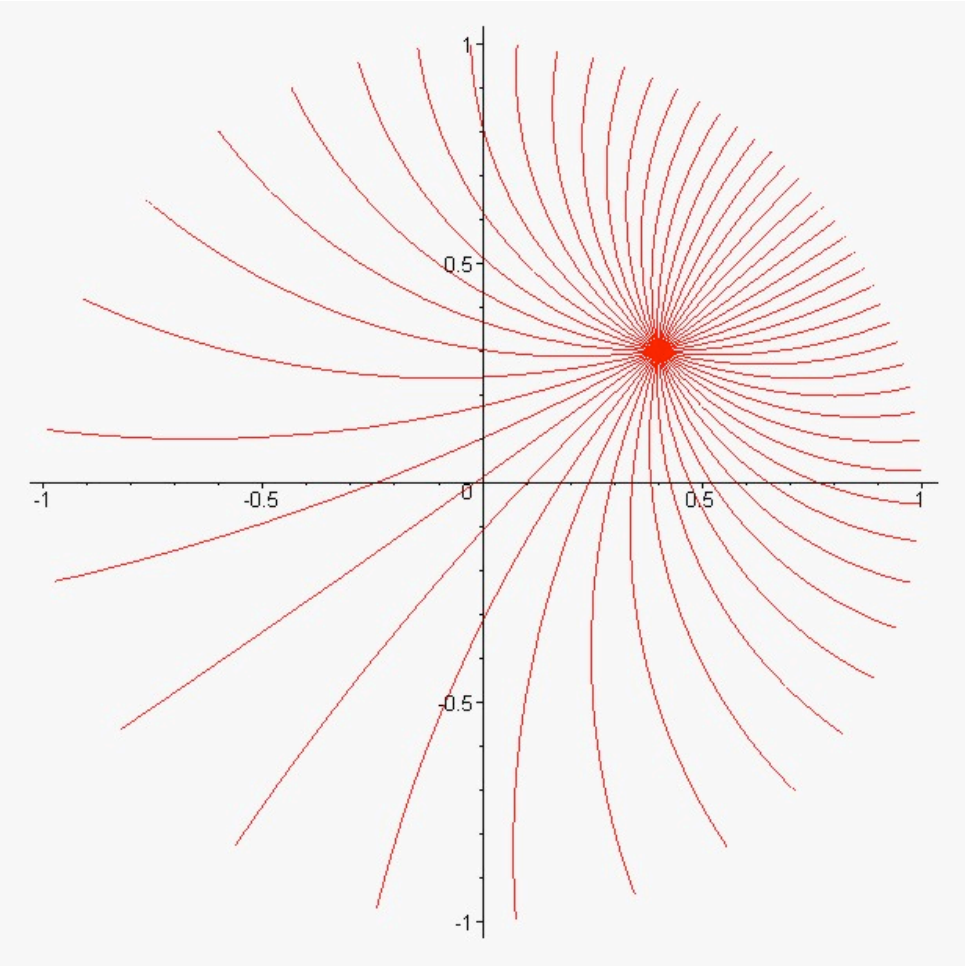


Figure 5.3: Geodesics emanating from a point in hyperbolic space. They all intersect the unit circle at right angles and each is in fact the arc of a circle. ($\lambda = 1$ has been taken here.)

or, equivalently,

$$\begin{aligned} \tanh^{-1}(|x'|) &= a\tanh^{-1}(|x|) \\ |x'| &= \tanh[a\tanh^{-1}(|x|)] \\ \frac{|x'|}{|x|} &= \tanh[a\tanh^{-1}(|x|)](|x|)^{-1}. \end{aligned}$$

We see that a hyperbolic dilation by a factor of a is therefore equivalent to an Euclidean dilation by $\tanh[a\tanh^{-1}(|x|)](|x|)^{-1}$.

Thus far, apart from the formula for non-Euclidean distance in terms of $X \cdot Y$, we have only mirrored what is already known. We now start to show the power of the conformal GA approach to this geometry, by showing how the operations and objects defined previously in conformal Euclidean geometry have immediate analogues here, representing a considerable unification and saving of effort. It may also be worth noting here that the power of this approach extends immediately to three or more dimensions where the Poincaré disc becomes a sphere. On the other hand, to provide a computational scheme which is somewhat akin to our rotor formulation, Brannan *et al* [7] introduce complex coordinates to work in the Poincaré disc and use Mobius transformations in place of the rotors. These are effective computationally in two dimensions, but of course the complex variable apparatus does not extend at all to three dimensions. Also the conformal setup enabling points, lines, circles, spheres and planes to be integrated into one algebraic system does not exist in the complex variable approach, even in two dimensions.

5.2.1 Geometric Objects in Hyperbolic Geometry

In this section we shall develop representations for geometric objects within hyperbolic geometry as we did previously for Euclidean geometry.

Hyperbolic lines

Having dealt with distances between points, let us start with the next most fundamental objects — lines. In Euclidean space we have seen that these are given by $L = n \wedge A \wedge B$, where A and B are the representatives of any two points on the line. Rotor transformations are able to move lines around successfully because for either a rotation or translation, $Rn\tilde{R} = n$. Thus when we perform $RL\tilde{R}$ we end up with $n \wedge (RA\tilde{R}) \wedge (RB\tilde{R})$ which is a line through the transformed points. Dilations also fit into this, since although they introduce a scale factor when acting on both n and general points, this still produces the intended line up to a scale factor in the Euclidean case.

This discussion makes it clear what a line must be in the conformal approach to hyperbolic geometry. Instead of using n, we must use the invariant object e. Thus we define a hyperbolic line as

$$L = e \wedge A \wedge B \tag{5.29}$$

where A and B are the two points through which we wish it to pass. This construction will guarantee covariance of the definition of a line, for the same reasons as in the Euclidean case (namely here that $Re\tilde{R} = e$ for any allowable rotor R).

We are faced with the immediate, very important question of what precisely *is* the object we have constructed. Ideally it should correspond to the d-line geodesics we have just discussed. To determine whether a position X lies on this line, we need to solve $X \wedge L = 0$. Let us take $x = x_1 e_1 + x_2 e_2$, $a = a_1 e_1 + a_2 e_2$ and $b = b_1 e_1 + b_2 e_2$. Then, taking $\lambda = 1$ for convenience, it is easy to show the resulting equation for x is of the form

$$x_1^2 + x_2^2 - 2px_1 - 2qx_2 + 1 = 0 \tag{5.30}$$

where $p^2 + q^2 > 1$. Specifically, one finds

$$p = -\frac{1}{2}\frac{\{(a_1{}^2 + a_2{}^2 + 1)\,b_2 - (b_1{}^2 + b_2{}^2 + 1)\,a_2\}}{a_1 b_2 - a_2 b_1}$$
$$q = -\frac{1}{2}\frac{\{-(a_1{}^2 + a_2{}^2 + 1)\,b_1 + (b_1{}^2 + b_2{}^2 + 1)\,a_1\}}{a_1 b_2 - a_2 b_1}$$

Equation 5.30 is precisely the form of equation given for d-lines in Brannan *et al*[7] page 283 and shows that indeed our recipe in terms of wedging with e has worked.

We can combine the notions of lines and distance, by asking for the 'hyperbolic midpoint' of the line segment joining two positions a and b. This is that point lying on the d-line between a and b which is an equal hyperbolic distance from each. Brannan *et al*[7] deal with this on page 288, but consider only the easiest case where the two points lie along a diameter of the unit disc.

We know, in the Euclidean case that forming $A + B$ will give a non-null point consisting of a multiple of the null vector representing the desired midpoint, plus a multiple of the point at infinity n. We hypothesise that we will

get the same behaviour here, but with e playing the role of n. We can use two different methods to obtain these results. The second is faster than the first, but we outline both, since they both typify the techniques available for use and serve as good illustrations of the power of the covariant method for hyperbolic geometry.

Firstly, since we can move objects around at will, let us establish the result in the simplest case – where the two points are symmetrically disposed about the origin, e.g. let $a = \alpha e_1$ and $b = -\alpha e_1$. Clearly, by symmetry the hyperbolic midpoint must be the origin itself, so we write

$$A + B = \beta(-\bar{n}) + \delta e \tag{5.31}$$

where δ and β are scalar multiples to be determined. Since $A+B = \frac{2}{\lambda^2-\alpha^2}(\alpha^2 n - \lambda^2\bar{n})$ it is easy to see that we must have

$$\delta = \frac{4\alpha^2}{\lambda^2-\alpha^2} \quad \text{and} \quad \beta = \frac{2(\alpha^2+\lambda^2)}{\lambda^2-\alpha^2} \tag{5.32}$$

Meanwhile, we note that

$$A \cdot B = -\frac{8\alpha^2\lambda^2}{(\lambda^2-\alpha^2)^2} \tag{5.33}$$

and some straightforward manipulation then tells us that

$$\delta = \sqrt{4-2A\cdot B} - 2 \quad \text{and} \quad \beta = \sqrt{4-2A\cdot B}. \tag{5.34}$$

We may solve the more general problem by rearranging equation 5.31 to get

$$-\bar{n} = \frac{1}{\sqrt{1-\frac{1}{2}A\cdot B}}\left\{\frac{1}{2}(A+B) - e\left(\sqrt{1-\frac{1}{2}A\cdot B} - 1\right)\right\} \tag{5.35}$$

Everything on the right hand side is covariant, and the separation between A and B is controlled by a variable which has been kept general (α), so by employing translation and rotation rotors on the right hand side we can make A and B line up with any two desired points. Meanwhile the left hand side will keep track, and must still remain the midpoint. To see this, note that its 'dot' with the new points that A and B transform into will remain constant during this process. The e term just remains invariant. For two completely general points A and B, therefore, their hyperbolic midpoint is given by

$$X_{\text{mid}} = \frac{1}{\sqrt{1-\frac{1}{2}A\cdot B}}\left\{\frac{1}{2}(A+B) - e\left(\sqrt{1-\frac{1}{2}A\cdot B}-1\right)\right\} \tag{5.36}$$

which is a fully covariant expression.

The alternative method, which is easier computationally, is as follows. Let us write equation 5.31 again but this time with X_{mid} in place of $-\bar{n}$ and with a general A and B in place from the beginning. So

$$A + B = \beta X_{\text{mid}} + \delta e \tag{5.37}$$

Rearranging, we have

$$X_{\text{mid}} = \frac{1}{\beta}(A + B - \delta e) \tag{5.38}$$

which shows that our assumption about the form of the midpoint is in fact valid. This is because, by dotting the right hand side with A and B in turn, we obtain

$$X_{\text{mid}} \cdot A = X_{\text{mid}} \cdot B = \frac{1}{\beta}(A \cdot B + \delta) \tag{5.39}$$

This means that X_{mid} is indeed equidistant, in a hyperbolic sense, from A and

B. Moreover,

$$X_{\text{mid}} \wedge (e \wedge A \wedge B) = 0 \tag{5.40}$$

so it correctly lies on the d-line joining them.

It just remains to fix δ and β by requiring that X_{mid} is null and is correctly normalised. The null requirement gives

$$2A \cdot B + 4\delta + \delta^2 = 0 \tag{5.41}$$

and requiring $X_{\text{mid}} \cdot e = -1$ yields

$$\beta = \delta + 2 \tag{5.42}$$

Solving these yields equation 5.34 as before, and we recover equation 5.36.

Hyperbolic circles

A great deal of hyperbolic geometry is concerned with hyperbolic circles, i.e. the locus of points that are at a constant hyperbolic distance from a given centre. We can immediately hypothesise that such a circle, passing through the points A, B, D should be given by the trivector.

$$C = A \wedge B \wedge D \tag{5.43}$$

The set of points X which satisfy $X \wedge C = 0$ can be found in a similar manner to the Euclidean case since the null-vector representation is the same up to a scale factor – i.e. in the Euclidean case we know that $X = \frac{1}{2\lambda^2}[x^2 n + 2\lambda x - \lambda^2 \bar{n}]$ and in the hyperbolic case $X = \frac{1}{\lambda^2 - x^2}[x^2 n + 2\lambda x - \lambda^2 \bar{n}]$. If the above hypothesis is true, we can immediately deduce the somewhat surprising – though in

fact true – conclusion that hyperbolic circles are also Euclidean circles. It turns out that it is just their centres that are, in general, different.

To establish that C is a hyperbolic circle, as well as clearly a Euclidean one, we can take the special case of a circle centred at the origin. By symmetry, this must be both a Euclidean and hyperbolic circle. Let this have Euclidean radius ρ. If we recall our earlier work, then for the scaled version of the null point representative, the inner product between two points A and B is given by

$$A \cdot B = -\frac{1}{2\lambda^2}(a-b)^2 \tag{5.44}$$

Then, since $X \cdot B = -\frac{1}{2\lambda^2}(x-b)^2 = -\frac{1}{2\lambda^2}(\rho)^2$ for a circle centre B and radius ρ, we have that

$$X \cdot (B - \frac{1}{2\lambda^2}(\rho)^2 n) = 0 \tag{5.45}$$

giving $C^* = B - \frac{1}{2\lambda^2}(\rho)^2 n$, where C is the trivector describing the circle. Thus if B is the origin, so that $B = -\frac{1}{2}\bar{n}$, the dual of C is given by $C^* = -\frac{1}{2\lambda^2}\{\rho^2 n + \lambda^2 \bar{n}\}$. Now, since $n + \bar{n} = 2e$, we can write $\rho^2 n + \lambda^2 \bar{n}$ as

$$\rho^2 n + \lambda^2 \bar{n} = 2\rho^2 e + (\lambda^2 - \rho^2)\bar{n} \tag{5.46}$$

We can normalise this to $C^2 = (C^*)^2 = 1$ via the scaling

$$C^* \mapsto \alpha(\rho^2 n + \lambda^2 \bar{n}) \quad \text{such that} \quad (C^*)^2 = 1$$

and hence $\alpha = \frac{1}{2\rho\lambda}$, thus

$$C^* = \frac{1}{2\rho\lambda}(\rho^2 n + \lambda^2 \bar{n}) = \frac{1}{2\rho\lambda}[2\rho^2 e + (\lambda^2 - \rho^2)\bar{n}]. \tag{5.47}$$

This displays the dual to C as a linear combination of the vector representing the origin, which is here the centre of the circle, and a multiple of e. This is the covariant form we require for generalisation. Note

$$X \cdot (IC) = \frac{1}{2\rho\lambda}\{X \cdot (2\rho^2 e + (\lambda^2 - \rho^2)\bar{n})\} = \frac{1}{2\rho\lambda}\{-2\rho^2 - (\lambda^2 - \rho^2)X \cdot (-\bar{n})\} \quad (5.48)$$

shows us how X maintaining a constant hyperbolic distance from the centre $(-\bar{n})$ is guaranteed by $X \cdot (IC) = 0$, in which case we have

$$X \cdot (-\bar{n}) = \frac{-2\rho^2}{(\lambda^2 - \rho^2)}. \quad (5.49)$$

Let us act on $IC = I(A \wedge B \wedge D)$ with hyperbolic rotors, to move the 3 points around as we wish. These same rotors acting on the right hand side of equation 5.46 mean that we will continue to get the required behaviour of constant hyperbolic distance from the transformed centre, since the rotors will take the origin to the new centre of the circle, say P. Thus indeed, the wedge of 3 points generates a hyperbolic circle, and moreover the above enables us to extract its (hyperbolic) radius and centre from the dual object. Again we see that the rôle of the n appearing in the Euclidean expressions is replaced by e here. Specifically we find the following: the (normalised dual to the) hyperbolic circle with hyperbolic centre P, Euclidean radius ρ and hyperbolic radius d is given by

$$IC = \frac{1}{2\rho\lambda}(2\rho^2 e + (\rho^2 - \lambda^2)P) \quad (5.50)$$

with $d = \sinh^{-1}(\rho/\sqrt{\lambda^2 - \rho^2})$, from equation 5.25.

Hyperbolic reflection

A very useful operation in hyperbolic geometry is the notion of hyperbolic reflection as defined, for example, in [7]. It is extremely easy to calculate the hyperbolic reflection of a point X in the d-line L in the GA approach. Assuming L is normalised to satisfy $L^2 = 1$, we just form LXL, the standard form of reflecting one object in another. Since $LeL = e$ for any normalised line, we see that $X' = LXL$ is both null and satisfies $X' \cdot e = -1$, thus qualifying it to represent a point. Moreover it is covariantly constructed; under a rotor transformation it just rotates to $RX'\tilde{R}$ and thus must represent something physical. It is not hard to show that the point it represents is that found by moving along a d-line, intersecting L at right angles and passing through X, by an equal hyperbolic distance on the other side of the line as X is on the original side. This is indeed the definition of reflection in this case.

5.2.2 Extension to Higher Dimensions and Other Geometries

All the above transfers seamlessly to three and higher dimensions, in most cases with no changes at all to the formulæ. This is in contrast to, for instance, the methods introduced in [7], which rely on standard complex analysis and so only work in the 2d plane. Furthermore, we can extend all the above analysis to the case of *spherical geometry* as well. This involves replacing the role of e with that of $\bar{e}$, which changes some trigonometric functions and ranges of applicability, but otherwise most of the above discussion goes through unchanged.

5.3 Non-Euclidean Visualisation Methods

A key requirement for visualising objects in the Poincaré disc representation of hyperbolic geometry is to plot representations of straight lines, known as d-lines. This section outlines a method developed to draw them using OpenGL and also presents a generalisation of the method for drawing analogous 'd-planes' in three-dimensional hyperbolic geometry.

5.3.1 NURBs

The d-lines on the Poincaré disc are circular arcs (and straight lines for the special cases of lines through the origin). OpenGL, the graphics library used for the implementation, has native support for a class of curves called Non-uniform Rational B-Splines (NURBS)[52].

NURBS curves are specified using a set of control points, P_i, weights, w_i and a set of normalised basis functions, $N_{i,k}$. The curve is given by

$$C(u) = \frac{\sum_{i=0}^{n} w_i P_i N_{i,k}(u)}{\sum_{i=0}^{n} w_i N_{i,k}(u)}$$

The basis functions are defined recursively.

$$N_{i,k}(u) = \frac{u - t_i}{t_{i+k} - t_i} N_{i,k-1}(u) + \frac{t_{i+k+1} - u}{t_{i+k+1} - t_{i+1}} N_{i+1,k-1}(u)$$

with

$$N_{i,0} = \begin{cases} 1 \text{ if } t_i \leq u \leq t_{i+1} \\ 0 \text{ otherwise} \end{cases}$$

and t_i being the elements of the *knot vector*

$$U = \{t_0, t_1, ..., t_m\}$$

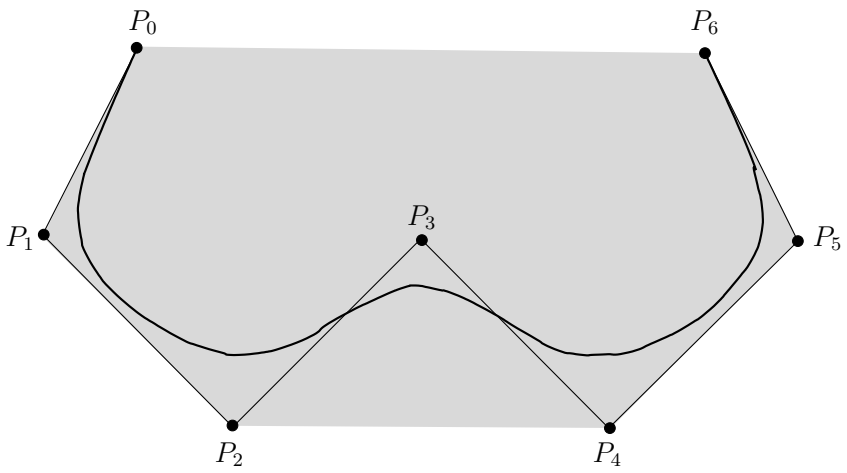


Figure 5.4: A set of control points and a typical example of an associated NURBS curve. Note that the endpoints of the curve are tangential to P_0P_1 and P_5P_6 and that the curve is within the convex hull of the points (shaded).

The relation between the number of knots, $m+1$, the degree k of the functions $N_{i,k}$ and the number of control points, $n+1$ is given by $m = n+k+1$ [51, 53].

Clearly a large family of curves can be expressed with suitable choices of knot vectors, weights and control points leading to great flexibility. All NURBS curves share some common properties, however, which make them useful in Computer Graphics. A NURBS curve *always* stays inside the convex hull of its control points [53] and thus it is straightforward to compute whether the curve will be displayed at all. Further they are tangential to the piece-wise linear interpolation of control points at the end-points as illustrated in figure 5.4.

5.3.2 Rendering *d*-lines

To draw *d*-lines on the Poincaré disc, we wish to draw circular arcs with end-points on the boundary circle and erupting normal to it.

A large number of different curves can be created with different control point numbers, positions and weights. Fortunately there are a number of standard techniques to generate common curves. One such method of drawing accurate circular arcs[59, 6] is useful to us. We use three control points: one at the start of the circular arc; one at the origin of the boundary circle; and one at its end (see figure 5.5). The end-point weights are unity whereas the weight of the control point at the origin is $\cos\gamma$, where γ is the angle OA makes with AB.

Although this procedure does generate a NURBS representation of a true circular arc the rendering of NURBS on a computer screen invariably involves a piecewise linear approximation and rasterization therefrom. This approximation can be made arbitrarily accurate to within machine precision and can certainly be made accurate to sub-pixel levels on any commonly used display.

Since NURBS are tangential to the piecewise linear interpolation of control points at either end, it is clear that the curve erupts from the boundary tangential to the lines OA and OB. Given O is the origin, and the boundary is centred upon it, these lines are radii of the boundary circle and so clearly are normals.

This allows us to construct a NURBS representation of any *d*-line on the

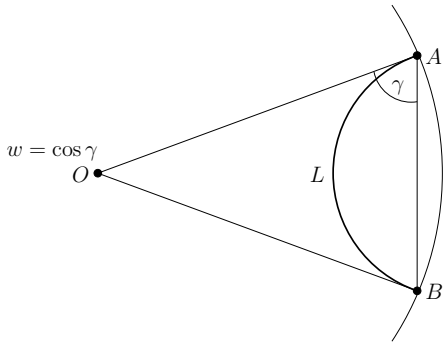


Figure 5.5: NURBS-based rendering of d-lines. Here O is the origin and A and B are the boundary points of the line L.

Poincaré disc (including diameters) and efficiently draw them using OpenGL.

Calculating properties of d-lines

The last step in drawing our d-lines is now finding where they intersect the boundary disc, their *boundary points*. Once we have these points, the drawing can be performed via our NURBS-based method outlined above.

We can calculate boundary points of d-lines by forming a circle corresponding to the boundary and finding the meet of the line with this circle. This gives the null-vector representation of the boundary points A and B as the bivector $A \wedge B$ as with circle/sphere intersections and the like. We have already shown that this bivector can be factorised into A and B via the method of projectors. We finally need to calculate the angle γ which is a trivial exercise in trigonometry.

Figure 5.6 shows the rendering of d-lines in action. Here we have three d-lines created by rotating and translating the diameter to form a hyperbolic

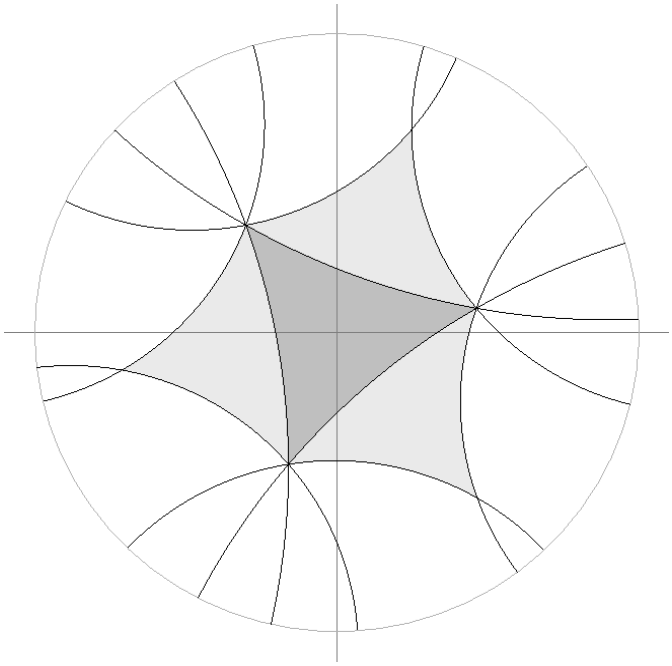


Figure 5.6: NURBS rendering of d-lines in action.

triangle (central dark-shaded region). Each of these lines was then reflected in the other two to form a set of three reflected triangles (outer light-shaded regions). This operation can be repeated to tile the space.

5.3.3 Rendering 'd-planes'

Drawing d-lines is interesting in itself but is an already solved problem many times over. What is interesting is the generality the GA approach provides. Recalling our discussion of the development of hyperbolic geometry in GA, at no point did we ever assume only two dimensions. We now have the intriguing opportunity to investigate visualisation algorithms in three dimensions.

We'll first assume that there is some analogous form to the Poincaré disc representation for three dimensions. In this case, d-lines can be drawn using a similar method, this time intersecting them with a boundary sphere to find the two points of eruption. What would be more interesting is attempting to find the form for 'd-planes'.

The method of defining planes in hyperbolic geometry is identical to our definition in Euclidean geometry; given four points on the d-plane, $\{x_1,...,x_4\}$, the plane Φ is defined as

$$\Phi = F_e(x_1) \wedge F_e(x_2) \wedge F_e(x_3) \wedge F_e(x_4) \tag{5.51}$$

Note we have incorporated the mapping into null-vectors within the definition. Any point, p, which lies on the plane Φ satisfies

$$F_e(p) \wedge \Phi = 0$$

The drawing of d-planes is, however, less straight forward. Firstly we need to find what shape they are when represented in the Poincaré sphere. Recall that

$$F_e(x) = \frac{1}{\lambda^2 - x^2}(x^2 + 2\lambda x - \lambda^2 \bar{n}) = \frac{2\lambda^2}{\lambda^2 - x^2} F(x)$$

where $F(x)$ is the mapping function for Euclidean geometry. The factor $(2\lambda^2)/(\lambda^2 - x^2)$ is always scalar for any vector x and we shall represent it as the function

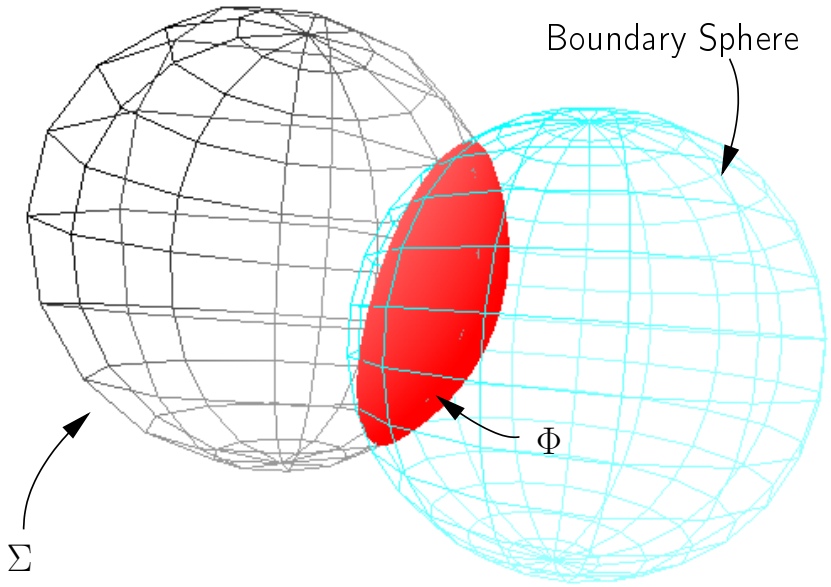


Figure 5.7: d-planes are caps of the corresponding Euclidean sphere.

$s(x)$. Hence we can re-write equation 5.51 as

$$\Phi = \bigwedge_{i=1\ldots4} s(x_i)F(x_i) \tag{5.52}$$

$$= \left[\prod_{i=1\ldots4} s(x_i)\right] \bigwedge_{i=1\ldots4} F(x_i) \tag{5.53}$$

$$= S(x_1,x_2,x_3,x_4) \bigwedge_{i=1\ldots4} F(x_i) \tag{5.54}$$

We have expressed Φ as the product of some scalar function, $S(\ldots)$ of the defining points and the *Euclidean* definition of a sphere. This allows us to infer that, within the Poincaré sphere, a d-plane passing through points $\{x_1,\ldots,x_4\}$ is represented by a sphere passing through those same points.

It has thus been found, without doing *any* explicit calculations with the

metric, that d-planes are represented in the Poincaré sphere by portions of spheres. This neatly shows the analytical simplicity that this approach provides. Figure 5.7 shows the relation between d-planes and the corresponding Euclidean sphere.

We can find the meet between the associated Euclidean sphere and the boundary sphere to give the circle of intersection. Inspecting figure 5.7 we see that this circle is the edge of the spherical cap corresponding to the d-plane.

The spherical cap forming the d-plane can be thought of as the intersection of the half-space containing the origin and bounded by the plane of intersection. The circle of intersection is important since we wish to extract this *plane* of intersection efficiently. If we note that the circle is equivalent to the wedge-product of three points on the circumference we can form the plane of the circle by simply wedging the circle with n. In summary, the plane of intersection, P can be found from the d-plane Φ in the following manner.

$$P = k(\Phi \vee B) \wedge n$$

where B is the Euclidean representation of the boundary sphere and k is some scale factor.

Bajaj *et al*[2] provide a method of finding a suitable set of control points and a NURBS parameter space clipping curve to draw spherical caps from sphere/half-space intersections. Their approach gives a set of control points and weights that together draw a little more than one hemisphere. Circular clipping paths in the parameter space are then used to form spherical caps.

This method was used to draw the spherical caps in the implementation.

5.4 Chapter summary

In this chapter we explored a method of extending the Conformal Model of Geometric Algebra to non-Euclidean geometries, specifically hyperbolic geometry. We outline a method for drawing d-lines and d-planes within a piece of visualisation software. In addition the power of the GA approach was demonstrated in that the basic building blocks of the Conformal Model, a mapping of vectors to null-vector representations and a set of rotors which act upon these null-vectors, are used in an identical manner to the Euclidean Conformal Model. This allows problems to be specified and solved in Euclidean geometry once, and by 'turning a knob' (replacing the null-vector mapping and rotors) the problem is solved for hyperbolic geometry. This is demonstrated in the next chapter, where we extend well known methods of generating complex-iteration based fractals to arbitrary dimension and in non-Euclidean geometries.

"For most of my life, one of the persons most baffled by my own work was myself."
— Benoît Mandelbrot

6

Generating Fractals using Geometric Algebra

In this chapter we investigate ways that a number of classical, well know fractals may be generated using Geometric Algebra. Further we show how GA can be used to form a natural generalisation to higher dimensions and well known fractals may be generated using Geometric Algebra. Further we show how GA can be used to form a natural generalisation to higher dimensions and non-Euclidean geometries. Some rendering strategies will also be discussed.

Fractals have always been a popular topic in computer graphics due to their ability to give rise to great æsthetic beauty from a relatively simple mathematical description. Generally a fractal is considered to be any geometric object which possesses detail on all scales[5, 41]. That is to say, one may examine the edge of the object under arbitrary magnification yet still find it rough and irregular. Many introductions to the subject of fractals and their creation on computers exist elsewhere[20, 31, 49].

The term fractal was coined by Mandelbrot[40] in 1975, originally from

the Latin *fractus* (broken) intended as a way of referring to their edges which looks like jagged cracks in some surface. Since many fractals (particularly those arising from so-caled *Iterated Function Systems*) have fractional Hausdorff dimension[21], some have joked that 'fractal' is actually a portmanteau word formed from 'fractionally' and 'dimensional'.

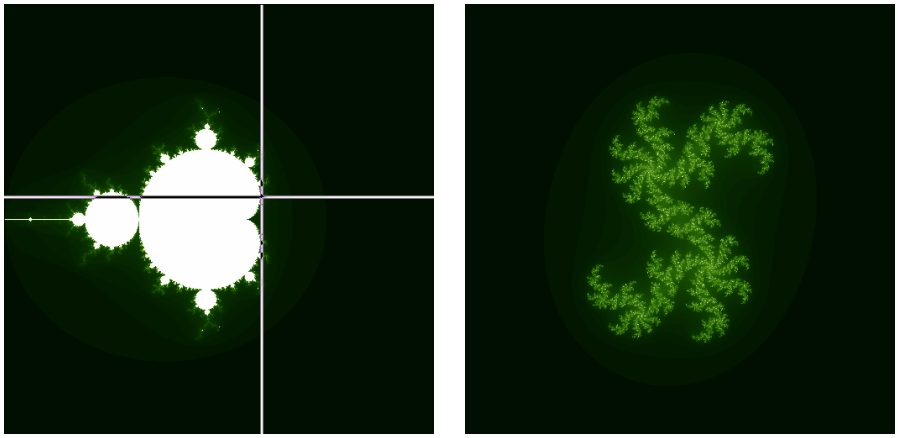
Below we shall investigate an extension, via GA, of the class of fractals which is most associated with Mandelbrot – those based on repeated iteration of a complex function. Similar fractals have found applications in a wide selection of research areas including image compression[4, 3].It is hoped that the GA-based approach here may also be of use in a similar manner however in this chapter we concentrate more on the æsthetic nature of the fractals.

6.1 Fractals from Complex Iteration

The classical Mandelbrot and Julia sets (figure 6.1) are well known, even to laymen, as examples of fractals. They are examples of a form of fractal known as *recurrence* or *escape-time* fractals. Such fractals are generated by iterating some complex function and noting how fast it 'escapes' to infinity (if at all).

Both fractals are displayed by mapping points in the complex plane to pixels in the final image. Usually this mapping to an image pixel location $\mathbf{x} = [\, x\, y\,]^T$ from its corresponding complex number, c, is simple.

Definition 6.1 (Representation of the Complex Plane)**.** An image point $\mathbf{x} = [\, x\, y\,]^T$ is mapped to a point on the complex plane via that mapping $\mathbf{x} \mapsto \mathcal{I}(x)$



(a) (b)

Figure 6.1: The well known (a) Mandelbrot set with the constant $c = 0.4 + 0.2i$ marked and (b) the Julia set associated with c.

where

$$\mathcal{J}(\mathbf{x}) = [\, a_1 x + b_1 \; a_2 y + b_2 \,]^T$$

and (b_1, b_2) specify the origin of the area of interest in the complex plane and (a_1, a_2) specify its scale.

The same complex function generates both the Mandelbrot and Julia sets[8, 16]. It is worth noting that other functions could be used and hence many other escape-time fractals exist. In this chapter, to save space, we shall only consider developments from this function.

Definition 6.2. The complex function $f(z)$, $c \in \mathbb{C}$, is defined as

$$f(z) = z^2 + c.$$

Definition 6.3 (Iteration)**.** Iteration of a function $f(x)$ is denoted as $f^{(n)}(z)$

where

$$f^{(n)}(z) \equiv f(f^{(n-1)}(z))$$

and $f^{(0)}(z) \equiv z$, $n \in \mathbb{Z}^+$.

6.1.1 The Mandelbrot Set

Definition 6.4 (The Mandelbrot set)**.** The Mandelbrot set, $\mathbb{M}$, is defined as

$$\mathbb{M} = \left\{ c \in \mathbb{C} : \lim_{n\to\infty} \left\| f^{(n)}(0) \right\| < \infty \right\}$$

where $\|z\| \equiv (zz^*)^{\frac{1}{2}}$.

Lemma 6.5. *If* $\left\| f^{(n)}(x) \right\| \geq 2$ *for some* n *and* x *then* $\left\| f^{(n)}(x) \right\| \to \infty$ *as* $n \to \infty$.

Proof. Suppose $\left\| f^{(n)}(x) \right\| > 2$ and $\left\| f^{(n)}(x) \right\| > \|c\|$. It is clear that

$$\frac{\left\| f^{(n+1)}(x) \right\|}{\left\| f^{(n)}(x) \right\|} = \frac{\left\| f^{(n)}(x)^2 + c \right\|}{\left\| f^{(n)}(x) \right\|}$$

and hence

$$\frac{\left\| f^{(n+1)}(x) \right\|}{\left\| f^{(n)}(x) \right\|} \geq \frac{\left\| f^{(n)}(x) \right\|^2 - \|c\|}{\left\| f^{(n)}(x) \right\|} = \left\| f^{(n)}(x) \right\| - \frac{\|c\|}{\left\| f^{(n)}(x) \right\|}.$$

As $\left\| f^{(n)}(x) \right\| > \|c\|$ and $\left\| f^{(n)}(x) \right\| > 2$ then

$$\left\| f^{(n)}(x) \right\| - \frac{\|c\|}{\left\| f^{(n)}(x) \right\|} > \left\| f^{(n)}(x) \right\| - 1 > 1$$

giving

$$\frac{\left\| f^{(n+1)}(x) \right\|}{\left\| f^{(n)}(x) \right\|} \geq 1$$

implying $\left\| f^{(n)}(x) \right\| \to \infty$ as $n \to \infty$ as required. □

1: $i_{\max} :=$ maximum number of iterations
2: **for all** points **x** in image **do**
3: $c := \mathcal{I}(\mathbf{x})$
4: $z := c$
5: $i := 0$
6: **while** $zz^* < 4$ and $i < i_{\max}$ **do**
7: $z := z^2 + c$
8: $i := i + 1$
9: **end while**
10: set pixel **x** to colour i
11: **end for**

Figure 6.2: Generating the Mandelbrot set

Using lemma 6.5 we may determine if some point x is *not* in the set as soon as $\left\| f^{(n)}(x) \right\| \geq 2$. In practise one may have to wait an arbitrarily long time for this condition to be met and one will never obtain it if x is within the set. We approximate the set by choosing some maximum number of iterations to wait before labelling x as being within the set. Our algorithm for generating an image of the set is shown in algorithm 6.2.

The level of detail of the resulting image being determined by the value of $i_{\max}$. The image of the Mandelbrot set in figure 6.1a was generated with $\mathbf{x} = [\, x\, y\,]^T, x \in (-2,2), y \in (-2,2)$. In figure 6.1a a colour palette was also chosen such that colour 0 was black and colour $i_{\max}$ was white moving through dark-green. The brightness of each pixel is therefore some measure of how long it took to decide whether that point was a member of the set.

Require: $c =$ constant
1: $i_{\max} :=$ maximum number of iterations
2: **for all** points **x** in image **do**
3: $z := \mathcal{I}(\mathbf{x})$
4: $i := 0$
5: **while** $zz^* < 4$ and $i < i_{\max}$ **do**
6: $z := z^2 + c$
7: $i := i + 1$
8: **end while**
9: set pixel x to colour i
10: **end for**

Figure 6.3: Generating the Julia set

6.1.2 The Julia Set

There are an infinite number of Julia sets; each point on the complex plane has a corresponding Julia set. The definition of the Julia set is somewhat similar to that of the Mandelbrot set.

Definition 6.6 (The Julia set)**.** The Julia set, $\mathbb{J}_c$, associated with the complex number c is given by

$$\mathbb{J}_c = \left\{ z \in \mathbb{C} : \lim_{n\to\infty} \left\| f^{(n)}(z) \right\| < \infty \right\}.$$

The difference between this and the Mandelbrot set is that there exists a Julia set associated with each complex number, c which must be chosen before generating the image. Our algorithm for generating Julia sets is, as one would expect, similar to that for the Mandelbrot set and is shown in algorithm 6.3.

Due to their similarity, there exist a number of theorems and conjectures

which link the Mandelbrot and Julia sets in some way. For example, if $c \in \mathbb{M}$ then the Julia set $\mathbb{J}_c$ is connected[1]. If c is near the border of $\mathbb{M}$ then it is a Cantor set[1].

6.2 Extending Complex Numbers

In this section we will seek to find a co-ordinate free analogue to the complex mapping $z \mapsto z^2$ in order to re-cast the fractals above in terms of geometric operations using GA. We will start by confining ourselves to the plane and then move into higher dimensions.

Firstly we define a mapping between the complex numbers, $\mathbb{C}$, and the vector-space of the complex plane, $\mathbb{R}^2$. Letting $\{e_1, e_2\}$ be some orthonormal basis for $\mathbb{R}^2$ we can form a natural one-to-one mapping between $r \in \mathbb{R}^2$ and $C(r) \in \mathbb{C}$:

Definition 6.7. Given some vector $r \in \mathbb{R}^2$, we can map one-to-one into the complex plane by forming the complex number

$$C(r) = (r \cdot e_1) + (r \cdot e_2)i.$$

Here it is clear that e_1 corresponds to the real-axis and e_2 corresponds to the imaginary axis of the complex plane. By squaring we have

$$\begin{aligned}[C(r)]^2 &= [(r \cdot e_1) + (r \cdot e_2)i]^2 \\ &= (r \cdot e_1)^2 - (r \cdot e_2)^2 + 2(r \cdot e_1)(r \cdot e_2)i.\end{aligned}$$

Lemma 6.8. *Given some vector $r \in \mathbb{R}^2$ representing the complex number $C(r)$, the mapping $r \mapsto re_1r$ is equivalent to $C(r) \mapsto [C(r)]^2$.*

Proof. Let $r = xe_1 + ye_2$. Therefore $C(r) = x + yi$. It is clear that

$$\begin{aligned} re_1 r &= (x^2 - y^2)e_1 + 2xye_2 \\ \Rightarrow\ C(re_1 r) &= (x^2 - y^2) + 2xyi \\ &= [C(r)]^2 \end{aligned}$$

as required. □

Complex addition is identical to vector addition using this representation.

Lemma 6.9. *Given vectors $r, c \in \mathbb{R}^2$ representing the complex numbers $C(r)$ and $C(c)$, the mapping $r \mapsto r + c$ is equivalent to $C(r) \mapsto C(r) + C(c)$.*

Proof. Clear by direct substitution. □

6.3 Moving to Higher Dimensions

In this section we extend the mapping above to more than two spatial dimensions. This turns out to be remarkably easy since the mapping is co-ordinate free; we simply remove the constraint that the vectors need be in $\mathbb{R}^2$.

Definition 6.10. Given a vector r and some unit basis vector e_1 the mapping $r \mapsto re_1 r$ is the geometric analogue of squaring a complex number.

Vector addition can once again be used in place of complex addition.

We may now define generalised Mandelbrot and Julia sets based upon a new vector recurrence relation.

Definition 6.11. The vector analogue of $f(r)$ is

$$f_v(r) = re_1 r + c$$

with initial values being defined by a particular fractal.

6.3.1 The Generalised Mandelbrot Set

We can now reformulate the definition of the Mandelbrot set in a co-ordinate free, dimension agnostic manner. Our new algorithm is shown in figure 6.4 and we may define the generalised Mandelbrot set similarly to the Mandelbrot set. Referring back to lemma 6.5 we see that the argument we used to terminate the iteration when $\left\| f^{(n)}(x) \right\| \geq 2$ still holds when we use $\|x\| = \sqrt{x \cdot x}$.

Definition 6.12 (The generalised Mandelbrot set)**.** The generalised Mandelbrot set, $\mathbb{M}_k$, in $\mathbb{R}^k$ is defined as

$$\mathbb{M}_k = \left\{ c \in \mathbb{R}^k : \lim_{n \to \infty} f_v^{(n)}(0) < \infty \right\}.$$

In figure 6.5 vectors lying in three orthogonal planes were used to generate three images of the three-dimensional Mandelbrot set which are then displayed mapped onto the original planes. This gives a crude visualisation method. A better method used to generate pictures of the generalised Julia set is given below.

The method used was to render a number of slices through the set leaving transparent the points in each slice which were not a part of the fractal. These slices were then stacked atop each other to give the impression of a three dimensional surface as illustrated in figure 6.8.

Require: Set $\mathcal{I}$ of vectors associated with image points
1: $i_{\max} :=$ maximum number of iterations
2: $e_1 :=$ a unit vector in some preferred direction
3: **for all** $c \in \mathcal{I}$ **do**
4: $\quad r := c$
5: $\quad i := 0$
6: $\quad$ **while** $r^2 < 4$ and $i < i_{\max}$ **do**
7: $\quad\quad r := re_1 r + c$
8: $\quad\quad i := i + 1$
9: $\quad$ **end while**
10: $\quad$ set pixel c to colour i
11: **end for**

Figure 6.4: Generating the Generalised Mandelbrot set

6.3.2 The Generalised Julia Set

We may generalise the Julia set in an analogous manner and generate it using algorithm 6.7.

Definition 6.13 (The generalised Julia set)**.** The generalised Julia set, $\mathbb{J}_{c,k}$, in $\mathbb{R}^k$ which is associated with the vector $c \in \mathbb{R}^k$ is given by

$$\mathbb{J}_{c,k} = \left\{ x \in \mathbb{R}^k : \lim_{n\to\infty} f_v^{(n)}(x) < \infty \right\}.$$

Figure 6.6 was rendered using a crude form of voxel rendering. A number of 2-dimensional 'slices' were rendered at varying heights in the set. Each pixel was coloured, or left transparent, depending on whether it was within or without the set. Finally, each slice was rendered at an oblique angle. The resulting 'stack' of slices gave an approximation to the true shape. Figure 6.8 illustrates this process.

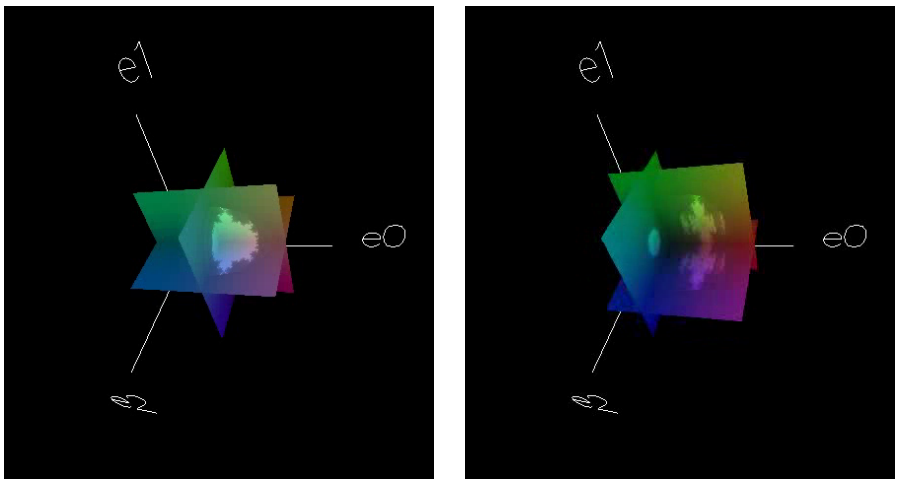


Figure 6.5: Two frames from an animation showing slices through the 3 dimensional Mandelbrot set.

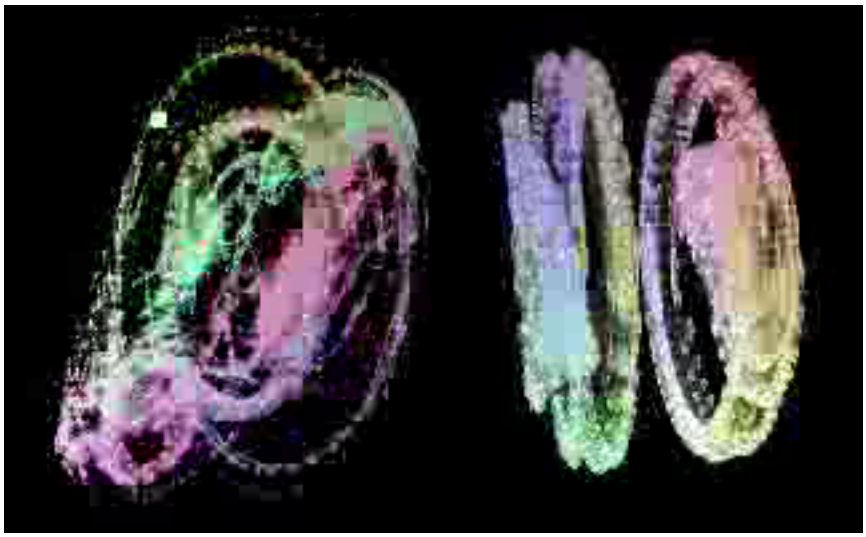


Figure 6.6: Two frames from an animation showing voxel rendering of 3d Julia sets.

Require: Set $\mathcal{I}$ of vectors associated with image points
Require: $c =$ constant vector
1: $i_{\max} :=$ maximum number of iterations
2: $e_1 :=$ a unit vector in some preferred direction
3: **for all** $r \in \mathcal{I}$ **do**
4: $\quad i := 0$
5: $\quad$ **while** $r^2 < 4$ and $i < i_{\max}$ **do**
6: $\quad\quad r := re_1 r + c$
7: $\quad\quad i := i + 1$
8: $\quad$ **end while**
9: $\quad$ set pixel r to colour i
10: **end for**

Figure 6.7: Generating the Generalised Julia set

The voxel rendering was adequate to visualise the fractals and confirm the algorithm works, but a more sophisticated rendering technique is desirable which can generate sharper images.

6.3.3 Ray Tracing

In this section we briefly extend the method in [14] of ray-tracing quaternionic escape-time fractals to the generalised GA fractals developed above. Ray-tracing of fractals is achieved by finding some distance function $d(x;\Omega)$ which gives the minimum distance to the fractal parameterised by Ω along the from the point x. For a Julia fractal the parameter Ω is the constant we have previously termed c. For a Mandelbrot fractal there is no parameter as the fractal is unique.

In [14] it was shown, for the complex number form of the escape time fractals above, that such a distance function at a point z in the complex plane

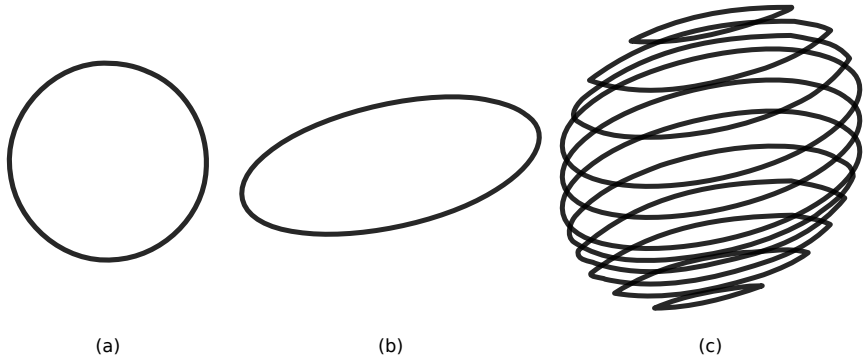


Figure 6.8: A crude form of voxel rendering. (a) A specific slice through the set. (b) Viewed from an oblique angle. (c) Stacked with other slices giving the impression of a three dimensional shape.

d_z was bounded by

$$d_z > \lim_{n\to\infty} \frac{|z_n|}{a} 2|z'_n| \log |z_n|$$

where $z_n = z_{n-1}^2 + c$, $|z'_n| = 2|z_{n+1}||z'_{n+1}|$, $z'_0 = 0$. The values of z_0 and c were dictated by the type of fractal as described above.

It was found that extending the formula to vectors using our analogue of complex multiplication yielded a suitable distance function although this has not yet been formally proved. Specifically

$$d(x;\Omega) = \lim_{n\to\infty} \frac{|x_n|}{a} 2|x'_n| \log |x_n|$$

where $x_n = f_v(x_{n-1})$, $|x'_n| = 2|x_{n+1}||x'_{n+1}|$, $x'_0 = 0$ and the initial value of x_n is fractal dependant.

Once a distance function (or a lower bound thereof) is available ray tracing becomes possible. Rays of light are traced back from point in the image plane of an imaginary camera to the scene. For a particular ray the algorithm to trace the fractal is as follows.

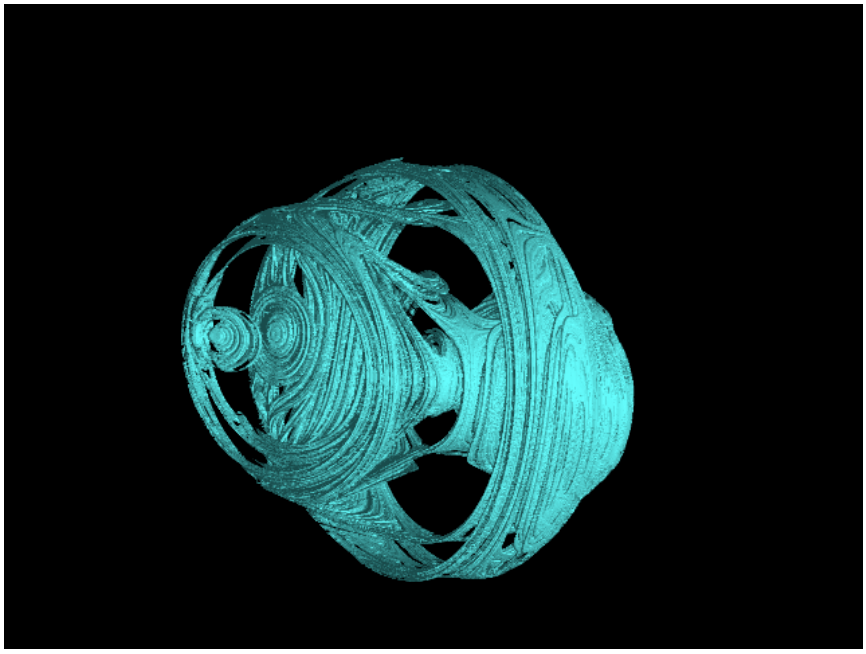


Figure 6.9: A ray-traced three-dimensional slice through a five-dimensional Julia set.

1. Set $\hat{r}$ to be a unit vector pointing along the ray direction.

2. Set the current position, x, to be the camera origin.

3. Calculate a lower bound, d_{-} for the distance from x.

4. If d_{-} is smaller than some tolerance τ exit reporting x as the intersection point with the fractal.

5. Set $x \leftarrow x + d_{-}\hat{r}$.

6. Go to step 3.

Once the intersection point is found the fractal may be lit by examining intersection points of neighbouring rays and taking the surface orientation

to be that of a plane passing through the neighbouring intersection points. Figure 6.9 shows a slice through a 5d Julia set rendered with this technique.

6.4 Moving to Hyperbolic Geometry

So far all our fractals have been generated using the Euclidean geometry of the complex plane or n-dimensional flat space. In this section we extend our algorithm using the non-Euclidean tools given to us by conformal GA. We firstly need to re-define our complex function in terms of the geometric operations.

Viewed using Euclidean geometry our function $f(r)$ firstly reflects and scales the vector $r \mapsto re_1 r$ and then translates via the vector c. We know already how to translate using conformal GA so we turn our attention to the former operation.

Since we constructed our GA approach to be analogous to the complex number approach we may borrow the geometric interpretation from complex numbers. In this case the mapping $r \mapsto r^2$ acts to rotate r by $\arg(r)$ and scale it by $\|r\|$. We may therefore extend this interpretation into our GA solution as in figure 6.10. We firstly work only in the $r \wedge e_1$ plane to allow our analogy with complex numbers to hold. If θ is the angle between r and e_1, i.e.

$$\theta = \cos^{-1}\left(\frac{r \cdot e_1}{\|r\|}\right),$$

then the mapping $r \mapsto re_1 r$ is initially a rotation in the plane $r \wedge e_1$ by θ followed by a dilation by a factor of $(r^2)^{\frac{1}{2}}$ which may be expressed in terms of

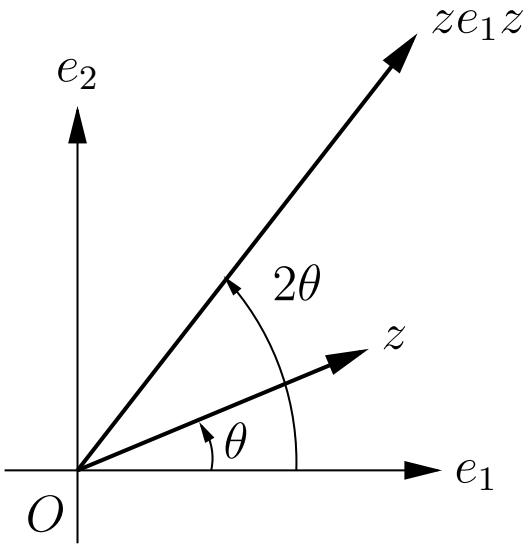


Figure 6.10: The geometrical interpretation of $r \mapsto re_1r$ as a rotation followed by a dilation.

rotors and dilators using their geometric definitions above.

Definition 6.14. The conformal GA analogue of $r \mapsto re_1r$ is given by

$$F(re_1r) = D_rR_rF(r)\tilde{R}_r\tilde{D}_r$$

where

$$R_r = \cos\frac{\theta}{2} + P\sin\frac{\theta}{2}, \quad P = \frac{(r \wedge e_1)}{\|r\|}$$

and θ is defined above. The dilator D_r acts to dilate about the origin by a factor of $\|r\|$. In Euclidean geometry it is given by

$$D_r = \exp\left(\frac{-\log(\|r\|)}{2}e\bar{e}\right).$$

The second part of $f(r)$ is a translation by c. A translator in non-Euclidean geometries is only defined as translating the origin to a given point so we

must be careful about the precise operations we perform. The GA analogue of the complex mapping $r \mapsto r + c$ is thus given by the following steps:

1. Translate r to the origin by applying the appropriate geometry-specific translator represented by $\tau(-r)$;
2. Translate the origin to c by applying the translator $\tau(c)$;
3. 'Undo' step 1 by applying the translator $\tau(r)$.

where $\tau(r)$ is a function which will give the appropriate translator for our chosen geometry. In Euclidean geometry $\tau(r) = 1 + \frac{nr}{2}$ and in non-Euclidean geometry

$$\tau(r) = \frac{1}{\sqrt{\lambda^2 - r^2}}(\lambda + \bar{e}r)$$

with λ being defined as in the discussion of non-Euclidean geometries above.

A little thought will reveal that this is equivalent to translating c by the translator $1 + \frac{nr}{2}$. We may therefore define our full conformal GA analogue of $f(r)$.

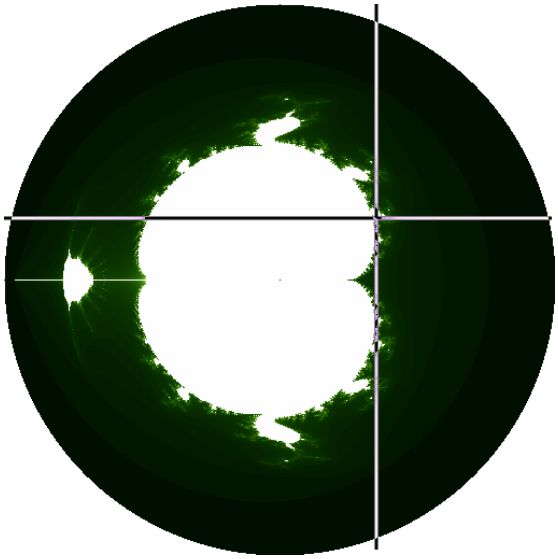
Definition 6.15. The conformal GA analogue of $f(r) = re_1 r + c$ is given by

$$f(r) = F^{-1}(T_r F(c) \tilde{T}_r)$$

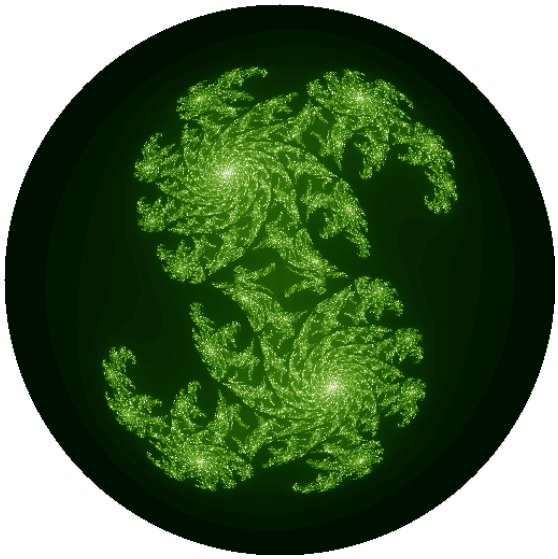
where

$$T_r = \tau(F^{-1}(D_r R_r F(r) \tilde{R}_r \tilde{D}_r))$$

Our algorithm for generating the generalised Mandelbrot and Julia sets is now identical except we substitute our new, geometric, definition of $f(r)$



(a)



(b)

Figure 6.11: The non-Euclidean analogue of the (a) Mandelbrot set with the constant $c = 0.4e_1 + 0.2e_2$ marked and (b) the Julia set associated with c.

and choose $\tau(r)$ and the form of D_r to reflect our chosen geometry. Usefully pure-rotation rotors remain the same in each geometry so no modification of them is necessary. Figure 6.11 shows a hyperbolic Mandelbrot and Julia set on the Poincaré disc generated with this method.

6.4.1 The Hyperbolic Mandelbrot Set

The hyperbolic Mandelbrot set is shown in figure 6.11a and is generated using algorithm 6.4, substituting step 7 for one performing the iteration outlined above.

6.4.2 The Hyperbolic Julia Set

A particular hyperbolic Julia set is shown in figures 6.11b, 6.12 and 6.13. Once again, it is generated using algorithm 6.7 substituting step 6. The two related montage figures show the same path across the underlying Mandelbrot set but with two differing methods of representing $x \mapsto x + c$. It is interesting to note that not only are the fractals different but they have different overall shape and behaviour. It is also worth remarking that, geometrically, there is no preferred form of the translation representation and so both these montages are equally valid images of hyperbolic Julia sets. This leads to the interesting conclusion that, whilst there is only one family of Euclidean Julia sets, for transitionally non-commuting geometries there are two.

It is worth noting that the code to generate the Euclidean fractals in figure 6.1 is identical to that used to generate 6.12 except for the definition of

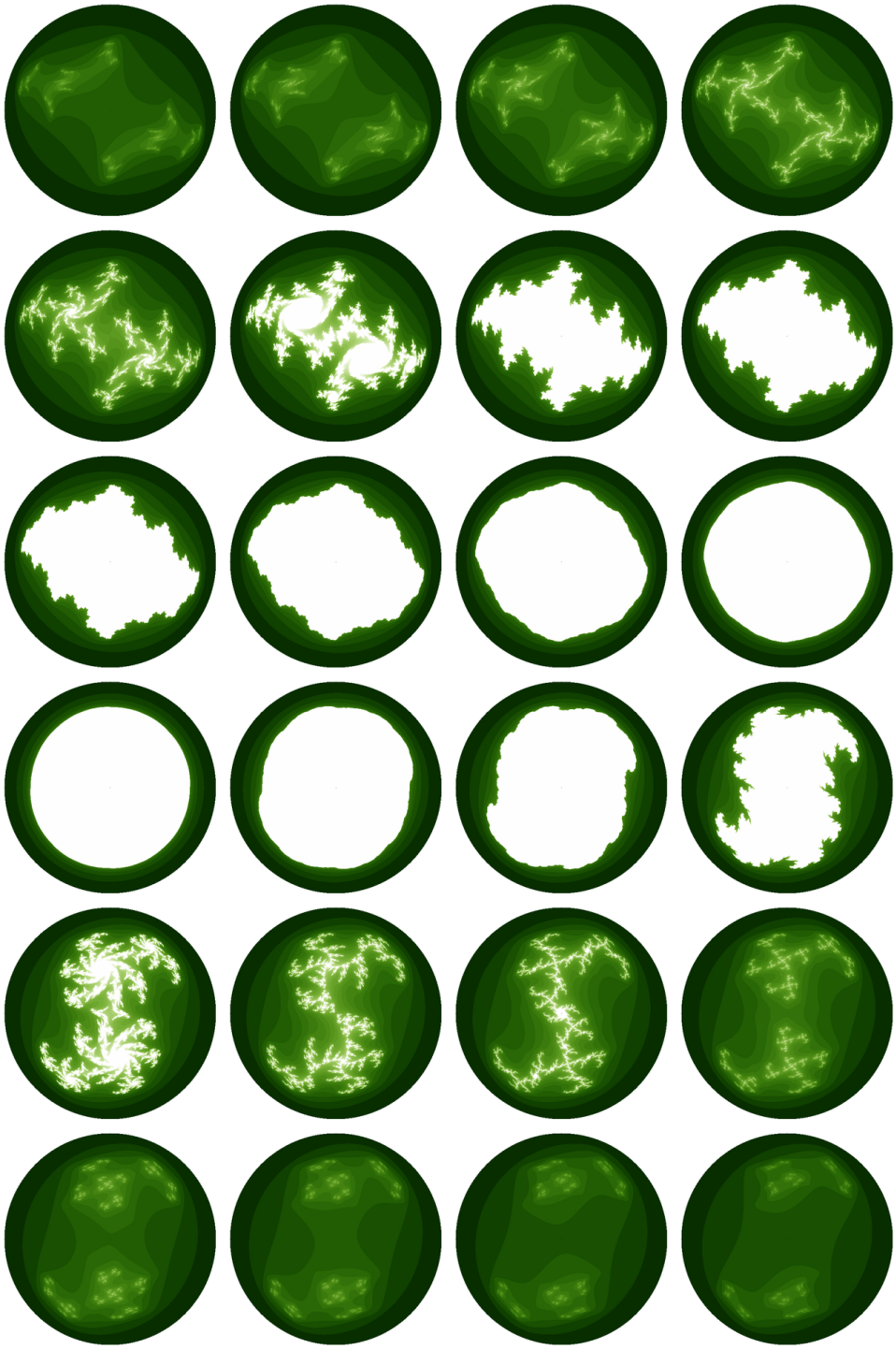


Figure 6.12: A montage of hyperbolic Julia sets where the constant c moves from $-0.7e_1 - 0.7e_2$ to $0.7e_1 + 0.7e_2$. In this figure translation $x \mapsto x + c$ is performed by applying a translation rotor corresponding to c to the vector x.

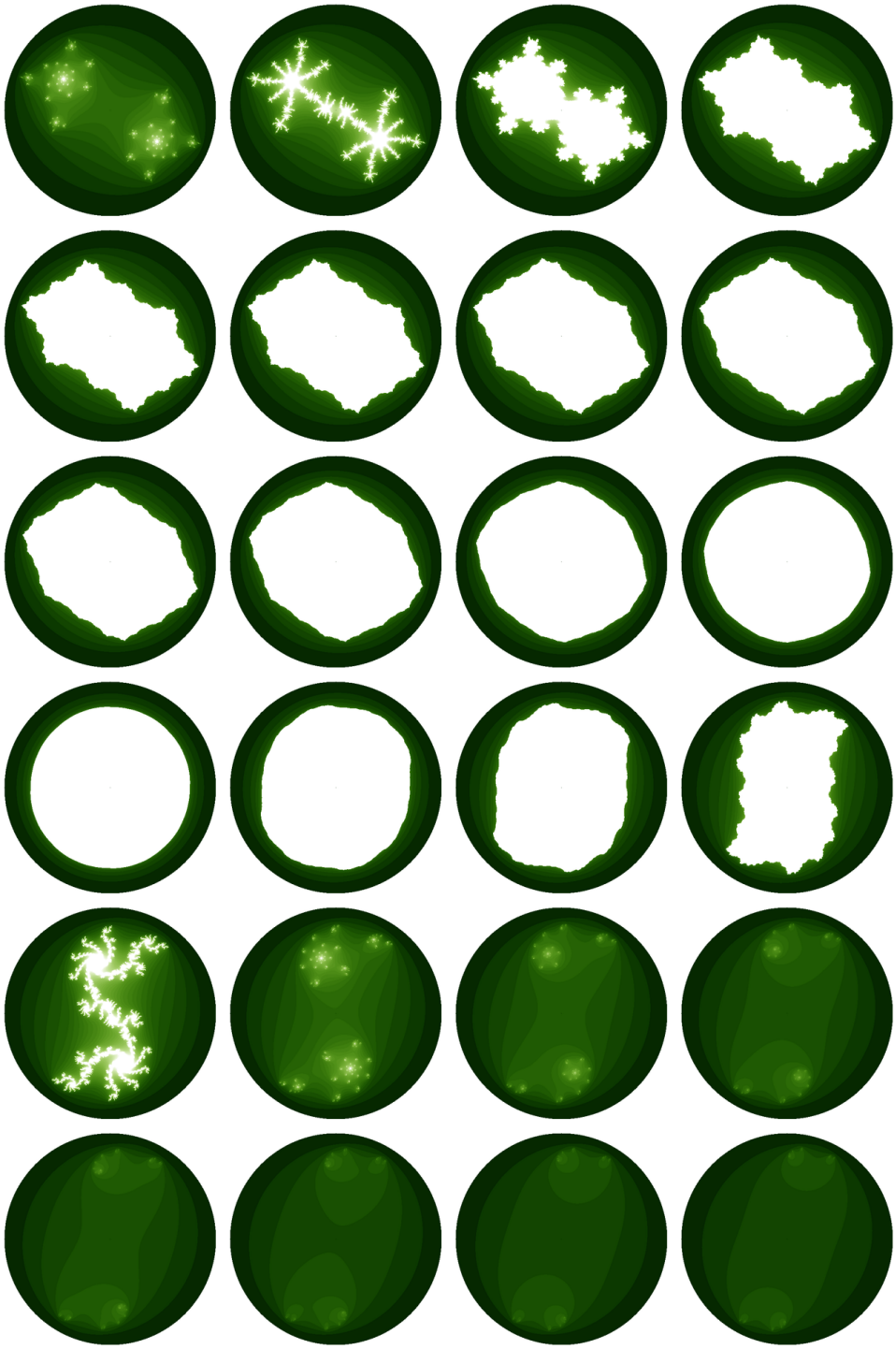


Figure 6.13: A montage of hyperbolic Julia sets where the constant c moves from $-0.7e_1 - 0.7e_2$ to $0.7e_1 + 0.7e_2$. In this figure translation $x \mapsto x + c$ is performed by applying a translation rotor corresponding to x to the vector c.

translators and dilators. We can replace the Euclidean form with those given in section 5.2 and re-cast the fractals into hyperbolic geometry. The fact that conformal GA algorithms developed using Euclidean geometry may so readily be applied to non-Euclidean geometries by simply changing the rotor definitions neatly indicates the power of the conformal GA approach.

"There is no branch of mathematics, however abstract, which may not someday be applied to the phenomena of the real world."
— Nicolai Lobachevsky

7

Rotors as Exponentiated Bivectors

In this chapter we shall consider common representations for pure-rotation rotors, translators and combinations of both. We shall term rotors which represent rotations and translations *general displacement* rotors. We will not explicitly deal with other rotors, such as dilation rotors, but similar techniques may be useful for their investigation. If we examine the form of the rotors presented in chapter 2 we see that all of them have a common form; they are all exponentiated bivectors. Rotations are generated by exponentiating bivectors with no e or $\bar{e}$-like components in them. We term objects with no contribution from e and $\bar{e}$ *spatial*. We may postulate that all general displacement rotors can be expressed as

$$R = \exp(B)$$

where B is the sum of two bivectors, one spatial and the other formed from the outer product of n with a spatial vector. Our justification is that those closely match the form of the bivectors which generate rotation and transla-

tion rotors respectively. The effect of this is to separate the basis bivectors of B into one with components of the form $e_i \wedge e_j$ and one with components of the form $e_i \wedge e$ and $e_i \wedge \bar{e}$.

The representation of rotors as generalised exponentials was explored by Rosenhahn *et al.*[57, 56, 55]. In this publication screw motions as the commuting product of a rotation and a translation out of the plane of rotation were viewed as a single exponentiated bivector. The set of screw-generating bivectors occupied a convenient sub-space of all bivectors. In this chapter we extend their work to exponentiate a wider class of bivector and, crucially, provide a closed form for the inverse operation. Later we show how this wider representation can be used to move to and from a 4×4 matrix representation of rotation and translation allowing use to 'glue' our GS-based methods to hardware especially designed for such matrix operations.

Notationally, we assume all general displacement rotors can be formed by exponentiating a bivector of the form $B = ab + cn$ where a, b and c are independent of n, i.e. if $n \in \mathcal{A}(m+1,1)$ then $\{a,b,c\} \in \mathbb{R}^m$. It is clear that the set of all B is some linear sub-space of all the bivectors.

We now suppose that we may interpolate rotors by defining some function $\ell(R)$ which acts upon rotors to give the generating bivector element. We then perform direct interpolation of these generators. We postulate that direct interpolation of these bivectors, as in the reformulation of quaternionic interpolation in section 2.1.3, will give some smooth interpolation between

the displacements. It is therefore a defining property of $\ell(R)$ that

$$R \equiv \exp(\ell(R)) \tag{7.1}$$

and so $\ell(R)$ may be considered as to act as a logarithm-like function in this context. It is worth noting that $\ell(R)$ does not possess all the properties usually associated with logarithms, notably, since $\exp(A)\exp(B)$ is not generally equal to $\exp(B)\exp(A)$ in non-commuting algebras, $\ell(\exp(A)\exp(B))$ cannot be equal to $A+B$ except in special cases.

To avoid the the risk of assigning more properties to $\ell(R)$ than we have shown, we shall resist the temptation to denote the function $\log(R)$. The most obvious property of $\log(\cdot)$ that $\ell(\cdot)$ does not possess is $\log(AB) = \log(A) + \log(B)$. This is clear since the geometric product is not commutative in general whereas addition is.

7.1 Form of $\exp(B)$ in Euclidean space

Definition 7.1 (Spatial elements)**.** An element x in the conformal model is termed 'spatial' if

$$x \cdot e = x \cdot \bar{e} = 0.$$

Lemma 7.2. *If B is of the form $B = \phi P + tn$ where t is a spatial vector, ϕ is some scalar and P is a spatial bivector where $P^2 = -1$ then, for any $k \in \mathbb{Z}^+$,*

$$B^k = \phi^k P^k + \alpha_k^{(1)} \phi P t n + \alpha_k^{(2)} \phi^2 P t n P + \alpha_k^{(3)} \phi t n P + \alpha_k^{(4)} t n$$

with the following recurrence relations for $\alpha_k^{(\cdot)}$, $k > 0$

$$\alpha_k^{(1)} = -\phi^2\alpha_{k-1}^{(2)} \quad \alpha_k^{(2)} = \alpha_{k-1}^{(1)}$$
$$\alpha_k^{(3)} = \alpha_{k-1}^{(4)} \quad \alpha_k^{(4)} = \phi^{k-1}P^{k-1} - \phi^2\alpha_{k-1}^{(3)}$$

with $\alpha_0^{(1)} = \alpha_0^{(2)} = \alpha_0^{(3)} = \alpha_0^{(4)} = 0$.

Proof. Firstly note that the theorem is provable by direct substitution for the cases $k = 0$ and $k = 1$. We thereafter seek a proof by induction.

Assuming the expression for B^{k-1} is correct, we post-multiply by $\phi P + tn$ to obtain

$$\begin{aligned} B^k &= \phi^k P^k + \alpha_{k-1}^{(1)}\phi^2 PtnP + \alpha_{k-1}^{(2)}\phi^3 PtnP^2 + \alpha_{k-1}^{(3)}\phi^2 tnP^2 + \\ &\quad \alpha_{k-1}^{(4)}\phi tnP + \phi^{k-1}P^{k-1}tn + \alpha_{k-1}^{(1)}\phi P(tn)^2 + \alpha_{k-1}^{(2)}\phi^2 PtnPtn + \\ &\quad \alpha_{k-1}^{(3)}\phi tnPtn + \alpha_{k-1}^{(4)}(tn)^2 \end{aligned}$$

Substituting $P^2 = -1$, $(tn)^2 = -tn^2t = 0$ and noting that $nPt = -Ptn$ leads to $tnPtn = -tPtn^2 = 0$

$$\begin{aligned} B^k &= \phi^k P^k + \alpha_{k-1}^{(1)}\phi^2 PtnP - \alpha_{k-1}^{(2)}\phi^3 Ptn - \alpha_{k-1}^{(3)}\phi^2 tn + \\ &\quad \alpha_{k-1}^{(4)}\phi tnP + \phi^{k-1}P^{k-1}tn \\ &= \phi^k P^k - (\alpha_{k-1}^{(2)}\phi^2)\phi Ptn + \alpha_{k-1}^{(1)}\phi^2 PtnP + \\ &\quad \alpha_{k-1}^{(4)}\phi tnP + (\phi^{k-1}P^{k-1} - \alpha_{k-1}^{(3)}\phi^2)tn \end{aligned}$$

Equating like coefficients we obtain the required recurrence relations. □

Lemma 7.3. *Assuming the form of B given in lemma 7.2, for* $k \in \mathbb{Z}^+$,

$$B^{2k} = (-1)^k\phi^{2k} - k(-1)^k\phi^{2k-1}[tnP + Ptn]$$

and

$$B^{2k+1} = (-1)^k \phi^{2k+1} P + k\phi^{2k}(-1)^k [tn - PtnP] + (-1)^k \phi^{2k} tn$$

Proof. Starting from $\alpha_0^{(\cdot)} = 0$ it is clear that the recurrence relations above imply that $\alpha_k^{(1)} = \alpha_k^{(2)} = 0 \; \forall k \geq 0$. Substituting $\alpha_k^{(3)} = \alpha_{k-1}^{(4)}$, that the relation for $\alpha_k^{(4)}$ is satisfied by

$$\alpha_k^{(4)} = \begin{cases} \frac{k}{2}(\phi P)^{k-1} & k \text{ even}, \\ \frac{k+1}{2}(\phi P)^{k-1} & k \text{ odd}. \end{cases}$$

When substituted into the expression for B^k, we obtain the result stated above.

□

Theorem 7.4. *If B is a bivector of the form given in lemma 7.2 then, defining $t_\parallel$ as the component of t lying in the plane of P and $t_\perp = t - t_\parallel$,*

$$\exp(B) = [\cos(\phi) + \sin(\phi)P]\,[1 + t_\perp n] + \operatorname{sinc}(\phi) t_\parallel n$$

Proof. Consider the power series expansion of $\exp(B)$,

$$\exp(B) = \sum_{k=0}^{\infty} \frac{B^k}{k!} = \sum_{k=0}^{\infty} \left[\frac{B^{2k}}{(2k)!} + \frac{B^{2k+1}}{(2k+1)!} \right]$$

Substituting the expansion for B^{2k} and B^{2k+1} from lemma 7.3

$$\begin{aligned} \exp(B) = &\sum_{k=0}^{\infty} \left[\frac{(-1)^k \phi^{2k}}{(2k)!} - k\frac{(-1)^k \phi^{2k-1}}{(2k)!} (tnP + Ptn) \right] \\ &+ \sum_{k=0}^{\infty} \left[\frac{(-1)^k \phi^{2k}}{(2k+1)!} (\phi P + tn) + k\frac{(-1)^k \phi^{2k}}{(2k+1)!} (tn - PtnP) \right] \end{aligned}$$

We now substitute the following power-series representations

$$\cos(z) = \sum_{k=0}^{\infty} \frac{(-1)^k z^{2k}}{(2k)!} \qquad \operatorname{sinc}(z) = \sum_{k=0}^{\infty} \frac{(-1)^k z^{2k}}{(2k+1)!}$$

$$-z\sin(z) = \sum_{k=0}^{\infty} 2k \frac{(-1)^k z^{2k}}{(2k)!} \quad \cos(z) - \operatorname{sinc}(z) = \sum_{k=0}^{\infty} 2k \frac{(-1)^k z^{2k}}{(2k+1)!}$$

to obtain

$$\begin{aligned}\exp(B) =& \cos\phi + \sin(\phi)\frac{1}{2}(tnP + Ptn) + \operatorname{sinc}(\phi)(\phi P + tn) \\ &+ \frac{1}{2}\left[\cos(\phi) - \operatorname{sinc}(\phi)\right](tn - PtnP)\end{aligned}$$

By considering parallel and perpendicular components of t with respect to the plane of P we can verify that $tnP + Ptn = 2Pt_\perp n$ and $PtnP = (t_\parallel - t_\perp)n$ hence

$$\begin{aligned}\exp(B) =& \cos\phi + \sin(\phi)Pt_\perp n + \operatorname{sinc}(\phi)(\phi P + tn) + \left[\cos(\phi) - \operatorname{sinc}(\phi)\right]t_\perp n \\ =& \cos(\phi)\left[1 + t_\perp n\right] + \sin(\phi)P\left[1 + t_\perp n\right] + \operatorname{sinc}(\phi)t_\parallel n \\ =& \left[\cos(\phi) + \sin(\phi)P\right]\left[1 + t_\perp n\right] + \operatorname{sinc}(\phi)t_\parallel n\end{aligned}$$

as required. □

Definition 7.5. A *screw* is a rotor whose action is to rotate by ϕ in the plane of P whilst translating along a vector a perpendicular to the plane of P. It may therefore be defined by the rotor

$$\tau(\phi, P, a) = \left[\cos\left(\frac{\phi}{2}\right) + \sin\left(\frac{\phi}{2}\right)P\right]\left[1 + \frac{na}{2}\right]$$

where ϕ is a scalar, P is a spatial bivector normalised such that $P^2 = -1$ and a is some vector satisfying $a \cdot n = a \cdot P = 0$.

Lemma 7.6. *The exponentiation function may be re-expressed using a screw*

$$\exp\left(\frac{\phi}{2}P+\frac{tn}{2}\right)=\left[1+\operatorname{sinc}\left(\frac{\phi}{2}\right)\frac{t_{\parallel}n}{2}\tilde{\tau}(\phi,P,-t_{\perp})\right]\tau(\phi,P,-t_{\perp})$$

Proof. We firstly substitute our definition of a screw into our form for the exponential

$$\exp\left(\frac{\phi}{2}P+\frac{tn}{2}\right)=\tau(\phi,P,-t_{\perp})+\operatorname{sinc}\left(\frac{\phi}{2}\right)\frac{t_{\parallel}n}{2}. \tag{7.2}$$

Noting that, since screws are rotors, $\tau(\cdot)\tilde{\tau}(\cdot)=1$, it is then clear that the required expression is equivalent to this form of the exponential. □

Lemma 7.7. *The expression*

$$1+\operatorname{sinc}\left(\frac{\phi}{2}\right)\frac{t_{\parallel}n}{2}\tilde{\tau}(\phi,P,-t_{\perp})$$

is a rotor which acts to translate along a vector $t'_{\parallel}$ given by

$$t'_{\parallel}=-\operatorname{sinc}\left(\frac{\phi}{2}\right)t_{\parallel}\left(\cos\left(\frac{\phi}{2}\right)-\sin\left(\frac{\phi}{2}\right)P\right)$$

Proof. The expression above may be obtained by substituting for the screw in the initial expression and simplifying. It is clearly a vector since multiplying $t_{\parallel}$ on the left by P is just a rotation by $\pi/2$ in the plane of P. □

We have now developed the required theories and tools to discuss the action of the rotor

$$R=\exp\left(\frac{\phi}{2}P+\frac{tn}{2}\right)$$

It translates along a vector $t_{\perp}$ being the component of t which does not lie in the plane of P, rotates by ϕ in the plane of P and finally translates along $t'_{\parallel}$

which is given by

$$t'_{\parallel} = -\text{sinc}\left(\frac{\phi}{2}\right) t_{\parallel} \left(\cos\left(\frac{\phi}{2}\right) - \sin\left(\frac{\phi}{2}\right) P\right)$$

which is the component of t lying in the plane of P, rotated by $\phi/2$ in that plane.

7.2 Checking $\exp(B)$ is a rotor

It is sufficient to check that $\exp(B)$ satisfies the following properties of a rotor R.

$$R\tilde{R} = 1, \quad RnR\tilde{} = n$$

Theorem 7.8. *If $R = \exp(B)$ and B is a bivector of the form given in lemma 7.2 then $R\tilde{R} = 1$.*

Proof. Consider the screw form of $\exp(B)$ from equation 7.2

$$R = \exp(B) = \tau(\phi, P, -t_{\perp}) + \text{sinc}\left(\frac{\phi}{2}\right)\frac{t_{\parallel} n}{2}$$

and make use of our knowledge that $\tau(\phi, P, -t_{\perp})$ is a rotor. Hence,

$$\begin{aligned} R\tilde{R} &= \tau(\phi, P, -t_{\perp})\tilde{\tau}(\phi, P, -t_{\perp}) + \text{sinc}^2\left(\frac{\phi}{2}\right)\frac{t_{\parallel} n^2 t_{\parallel}}{4} \\ &\quad + \text{sinc}\left(\frac{\phi}{2}\right)\left[\tau(\phi, P, -t_{\perp}) n t_{\parallel} + t_{\parallel} n \tilde{\tau}(\phi, P, -t_{\perp})\right] \\ &= 1 + 0 + \text{sinc}\left(\frac{\phi}{2}\right)\left[T + \tilde{T}\right] \end{aligned}$$

where $T = \tau(\phi, P, -t_{\perp}) n t_{\parallel}$.

Looking at the definition of $\tau(\phi, P, -t_{\perp})$, it is clear that it has only scalar, bivector and 4-vector components, with the bivector components being parallel to P or $t_{\perp} n$ and the 4-vector components being parallel to $P t_{\perp} n$. When

post-multiplied by $nt_{\|}$ to form T, the 4-vector component goes to zero (since $n^2 = 0$) as does the bivector component parallel to $t_{\perp}n$ and so we are left with T having only components parallel to $nt_{\|}$ and $Pnt_{\|}$. We may now express T as

$$T = \alpha nt_{\|} + \beta Pnt_{\|}$$

where α and β are suitably valued scalars. Hence

$$T + \tilde{T} = \alpha \left[nt_{\|} + t_{\|}n\right] + \beta \left[Pnt_{\|} + t_{\|}n\tilde{P}\right] = 0 + \beta n \left[Pt_{\|} - t_{\|}\tilde{P}\right]$$

By considering two basis vectors of P, a and b, such that $P = ab$, $a \cdot b = 0$ and resolving $t_{\|}$ in terms of a and b it is easy to show that $Pt_{\|} - t_{\|}\tilde{P} = 0$ and hence $T + \tilde{T} = 0$ giving the required result. □

Theorem 7.9. *If $R = \exp(B)$ and B is a bivector of the form given in lemma 7.2 then $RnR = n$.*

Proof. Again using the screw form of R from equation 7.2 we have

$$\begin{aligned} Rn &= \tau(\phi, P, -t_{\perp})n + \operatorname{sinc}\left(\frac{\phi}{2}\right)\frac{t_{\|}n^2}{2} \\ &= \tau(\phi, P, -t_{\perp})n + 0 \end{aligned}$$

Defining the rotation rotor $R_{(P,\phi)}$ as

$$R_{(P,\phi)} = \cos\left(\frac{\phi}{2}\right) + \sin\left(\frac{\phi}{2}\right)P$$

and substituting for the definition of the screw above gives

$$Rn = R_{(P,\phi)}n$$

Similarly, again using the screw form of R we have

$$\begin{aligned} nR &= n\tau(\phi, P, -t_\perp) + \operatorname{sinc}\left(\frac{\phi}{2}\right)\frac{nt_\parallel n}{2} \\ &= n\tau(\phi, P, -t_\perp) + 0 \\ &= nR_{(P,\phi)}\left(1 + \frac{tn}{2}\right) \\ &= R_{(P,\phi)}n\left(1 + \frac{tn}{2}\right) \\ &= R_{(P,\phi)}n \end{aligned}$$

We now have that $Rn = nR$ and hence, using $R\tilde{R} = 1$ from the previous theorem, $Rn\tilde{R} = nR\tilde{R} = n$. □

7.3 Method for evaluating $\ell(R)$

We have found a form for $\exp(B)$ given that B is in a particular form. Now we seek a method to take an arbitrary displacement rotor $R = \exp(B)$ and re-construct the original B. Should there exist a B for all possible R, we will show that our initial assumption that all displacement rotors can be formed from a single exponentiated bivector of special form is valid. We shall term this initial bivector the *generator* rotor (to draw a parallel with Lie algebras).

We can obtain the following identities for $B = (\phi/2)P + tn/2$ by simply considering the grade of each component of the exponential

$$\begin{aligned} \langle R \rangle_0 &= \cos\left(\frac{\phi}{2}\right) \\ \langle R \rangle_2 &= \sin\left(\frac{\phi}{2}\right)P + \cos\left(\frac{\phi}{2}\right)t_\perp n + \operatorname{sinc}\left(\frac{\phi}{2}\right)t_\parallel n \\ \langle R \rangle_4 &= \sin\left(\frac{\phi}{2}\right)Pt_\perp n \end{aligned}$$

It is straightforward to reconstruct ϕ, $t_\perp$ and $t_\parallel$ from these components by partitioning a rotor as above. Once we have a method which gives the generator B for any displacement rotor R, we have validated our assumption.

Theorem 7.10. *The inverse-exponential function $\ell(R)$ is given by*

$$\ell(R) = ab + c_\perp n + c_\parallel n$$

where

$$\begin{aligned}
\|ab\| &= \sqrt{|(ab)^2|} = \cos^{-1}(\langle R\rangle_0) \\
ab &= \frac{(\langle R\rangle_2 n)\cdot e}{\operatorname{sinc}(\|ab\|)} \\
c_\perp n &= -\frac{ab\langle R\rangle_4}{\|ab\|^2 \operatorname{sinc}(\|ab\|)} \\
c_\parallel n &= -\frac{ab\langle ab\langle R\rangle_2\rangle_2}{\|ab\|^2 \operatorname{sinc}(\|ab\|)}
\end{aligned}$$

Proof. It is clear from the above that the form of $\|ab\|$ is correct. We thus proceed to show the remaining equations to be true

$$\begin{aligned}
\langle R\rangle_2 &= \cos(\|ab\|)\, c_\perp n + \operatorname{sinc}(\|ab\|)\left[ab + c_\parallel n\right] \\
\langle R\rangle_2 n &= \operatorname{sinc}(\|ab\|)\, abn \\
(\langle R\rangle_2 n)\cdot e &= \operatorname{sinc}(\|ab\|)\, ab
\end{aligned}$$

therefore the relation for ab is correct.

$$\begin{aligned}
\langle R\rangle_4 &= \operatorname{sinc}(\|ab\|)\, abc_\perp n \\
ab\langle R\rangle_4 &= -\|ab\|^2 \operatorname{sinc}(\|ab\|)\, c_\perp n
\end{aligned}$$

and hence the relation for $c_\perp n$ is correct.

$$\begin{aligned}
\langle R \rangle_2 &= \cos(\|ab\|)\, c_\perp n + \operatorname{sinc}(\|ab\|)\left[ab + c_\| n\right] \\
ab\langle R \rangle_2 &= \cos(\|ab\|)\, abc_\perp n + \operatorname{sinc}(\|ab\|)\left[abc_\| n - \|ab\|^2\right] \\
\langle ab\langle R \rangle_2 \rangle_2 &= \operatorname{sinc}(\|ab\|)\, abc_\| n
\end{aligned}$$

and so the relation for $c_\| n$ is correct. □

7.4 Mapping Generators to Matrices

Although the representation of a rotor as an exponentiated generator bivector is convenient mathematically and useful for smooth pose interpolation and mesh deformation, as will be presented later, it is somewhat cumbersome to integrate into an existing graphical pipeline. Most graphics hardware and nearly all graphics APIs represent rigid-body transformations as 4×4 matrices. Specifically, given the projective mapping from a three-dimensional vector $\mathbf{x}$ to its homogeneous representation,

$$\mathbf{x} \mapsto \begin{bmatrix} w\mathbf{x} \\ w \end{bmatrix}$$

where w is some arbitrary non-zero scalar, a rigid body transform can be represented as

$$\begin{bmatrix} w\mathbf{x} \\ w \end{bmatrix} \mapsto \mathcal{R} \begin{bmatrix} w\mathbf{x} \\ w \end{bmatrix}$$

and

$$\mathcal{R} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ 0 & 1 \end{bmatrix}.$$

Here **R** is an orthonormal 3×3 rotation matrix and **t** is some translation vector.

Due to the common nature of such a representation, it would be advantageous to have some method of mapping between the conveniently linear space of generators to the non-linear space of these 4×4 matrices. In this section we shall develop such a method. Note that we shall only be working in three dimensions due to the limitations of the matrix representation rather than the exponentiated generator representation.

Recall that we represent a rotor, R, as $R = \exp(B)$ where B is a bivector. The generator bivector, B, may itself be parametrised in terms of a spatial bivector P which is normalised such that $P^2 = -1$, a scalar ϕ and a spacial vector t as in lemma 7.2. Letting $P = p_1 e_{12} + p_2 e_{23} + p_3 e_{31}$ and $t = t_1 e_1 + t_2 e_2 + t_3 e_3$ we may represent B via the vector **b**

$$\mathbf{b} = [p_1 \; p_2 \; p_3 \; t_1 \; t_2 \; t_3]^T.$$

Using this generator representation is useful since any vector $\mathbf{b} \in \mathbb{R}^6$ will represent a valid generator and hence a valid rotor.

It is also worth noting that this method of converting from a matrix representation to a generator is ambiguous inasmuch as the matrix representation cannot uniquely represent rotations by more than 2π.

7.4.1 Method

We shall attempt to represent the application of a rotor associated with a generator to a point by defining an appropriate linear function $h(\cdot)$.

Definition 7.11. The function $h(\phi, P, t, p)$ is defined as

$$h(\phi, P, t, p) = F^{-1}\left(RF(p)\tilde{R}\right)$$

where $R = \exp(B)$, $B = \phi P + tn$ and P, ϕ and t are as defined in lemma 7.2. $F(\cdot)$ is the mapping of vectors to their null-vector representation.

It is easy, if somewhat tedious, to show by direct expansion and comparison of terms that an expression for $h(\cdot)$ which matches the definition above is

$$\begin{aligned} h(\phi, P, t, p) = {} & c^2 p - s^2 PpP - sc\,[pP - Pp] \\ & + \left[\frac{kc+1}{2}t + \frac{kc-1}{2}PtP - \frac{sk}{2}(tP - Pt)\right] \end{aligned} \tag{7.3}$$

where $s = \sin(\phi/2)$, $c = \cos(\phi/2)$ and $k = \operatorname{sinc}(\phi/2)$.

Definition 7.12. The function $\mathbf{v}(x)$ maps the m-dimensional vector x to its column-vector representation

$$\mathbf{v}(x) = \begin{bmatrix} x \cdot e_1 \\ x \cdot e_2 \\ \vdots \\ x \cdot e_m \end{bmatrix}.$$

It is clear by inspection that the function $h(\cdot)$ is constant with respect to p added to a function linear in p. We can therefore find some 3×4 matrix $\mathcal{H}(\phi, P, t)$, such that

$$\mathbf{e}^T \mathcal{H}(\phi, P, t) \begin{bmatrix} \mathbf{v}(p) \\ 1 \end{bmatrix} = h(\phi, P, t, p) \tag{7.4}$$

where $\mathbf{e}^T = [e_1\ e_2\ e_3]$.

Comparing the action of $\mathcal{H}$ and the form of $[\mathbf{v}(p)\ 1]^T$ to the discussion of 4×4 transformation matrices above, it is easy to see that the mapping $p \mapsto h(\phi, P, t, p, 1)$ is equivalent to

$$\begin{bmatrix} \mathbf{v}(p) \\ 1 \end{bmatrix} \mapsto \mathcal{R} \begin{bmatrix} \mathbf{v}(p) \\ 1 \end{bmatrix}$$

with

$$\mathcal{R} = \begin{bmatrix} & \mathcal{H} & \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Mapping a given generator parametrised in terms of ϕ, P and t therefore requires finding a closed form of $\mathcal{H}$ given $h(\cdot)$.

7.4.2 Finding $\mathcal{H}$ from a generator

In this section we shall seek a method of directly obtaining an appropriate form for $\mathcal{H}$ which represents the same action as a particular generator. Ideally this mapping should be simple enough to fit inside the programmable portions of a Graphics Processing Unit, allowing for hardware accelerated generator-based algorithms to be implemented on consumer-level graphics hardware.

Definition 7.13. Define the resolution of a bivector A onto the orthonormal basis set $\{e_{12}, e_{23}, \cdots\}$ as the set of scalars $\{a_{12}, a_{23}, \cdots\}$ where

$$A = a_{12}e_{12} + a_{23}e_{23} + \cdots.$$

Definition 7.14. Define the resolution of a vector b onto the orthonormal basis set $\{e_1, e_2, \cdots\}$ as the set of scalars $\{b_1, b_2, \cdots\}$ where

$$b = b_1 e_1 + b_2 e_2 + \cdots .$$

Definition 7.15. Define the function $\mathbf{f}_1(A, b)$, with A a three-dimensional bivector and b a three-dimensional vector, as

$$\mathbf{f}_1(A, b) = \begin{bmatrix} 0 & a_{12} & -a_{31} \\ -a_{12} & 0 & a_{23} \\ a_{31} & -a_{23} & 0 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}.$$

Definition 7.16. Define the function $f_2(A, b)$, with A a three-dimensional bivector and b a three-dimensional vector, as

$$f_2(A, b) = (a_{12} b_3 + a_{23} b_1 + a_{31} b_2).$$

Corollary 7.17. *Given A a three-dimensional bivector and b a three-dimensional vector*

$$bA - Ab = -[e_1\ e_2\ e_3]\, 2\mathbf{f}_1(A, b).$$

Proof. With the definitions for resolving A and b above one can show by direct expansion over an orthonormal basis that

$$Ab = f_2(A, b)\ e_{123} + [e_1\ e_2\ e_3]\, \mathbf{f}_1(A, b)$$

and

$$bA = f_2(A, b)\ e_{123} - [e_1\ e_2\ e_3]\, \mathbf{f}_1(A, b)$$

The desired result is then clear by substitution. □

Definition 7.18. Define $\mathbf{M}_1(A)$ to be a function of a three-dimensional bivector A,

$$\mathbf{M}_1(A) = \begin{bmatrix} 0 & a_{12} & -a_{31} \\ -a_{12} & 0 & a_{23} \\ a_{31} & -a_{23} & 0 \end{bmatrix}.$$

Corollary 7.19. *Equation 7.3 is equivalent to*

$$\begin{aligned} h(\phi, P, t, p, \lambda) = c^2\ \mathbf{e}^T\mathbf{v}(p) - s^2 PpP + 2sc\ \mathbf{e}^T\mathbf{M}_1(P)\mathbf{v}(p) \\ + \lambda\left[\frac{kc+1}{2}\mathbf{e}^T\mathbf{v}(t) + \frac{kc-1}{2}PtP + sk\ \mathbf{e}^T\mathbf{M}_1(P)\mathbf{v}(t)\right]. \end{aligned}$$

Proof. Direct substitution and application of lemma 7.17. □

Definition 7.20. Define $\mathbf{M}_2(A)$ to be a function of a three-dimensional bivector A,

$$\mathbf{M}_2(A) = \begin{bmatrix} a_{12}^2 - a_{23}^2 + a_{31}^2 & -2a_{23}a_{31} & -2a_{12}a_{23} \\ -2a_{23}a_{31} & a_{12}^2 + a_{23}^2 - a_{31}^2 & -2a_{12}a_{31} \\ -2a_{12}a_{23} & -2a_{12}a_{31} & -a_{12}^2 + a_{23}^2 + a_{31}^2 \end{bmatrix}.$$

Corollary 7.21. *Given a three-dimensional bivector A and a three-dimensional vector b,*

$$AbA = \mathbf{e}^T\mathbf{M}_2(A)\mathbf{v}(b).$$

Proof. Clear by substitution and expansion. □

Lemma 7.22. *An equivalent form for $h(\cdot)$ as given in equation 7.3 is*

$$\begin{aligned} h(\phi, P, t, p, \lambda) = \mathbf{e}^T\left[c^2\ \mathbf{v}(p) - s^2\ \mathbf{M}_2(P)\mathbf{v}(p) + 2sc\ \mathbf{M}_1(P)\mathbf{v}(p)\right] \\ + \lambda\mathbf{e}^T\left[\frac{kc+1}{2}\mathbf{v}(t) + \frac{kc-1}{2}\ \mathbf{M}_2(P)\mathbf{v}(t) + sk\ \mathbf{M}_1(P)\mathbf{v}(t)\right]. \end{aligned}$$

Proof. Substitute the above corollaries into $h(\phi, P, t, p, \lambda)$ to obtain

$$h(\phi,P,t,p,\lambda) = c^2\ \mathbf{e}^T\mathbf{v}(p) - s^2\ \mathbf{e}^T\mathbf{M}_2(P)\mathbf{v}(p) + 2sc\ \mathbf{e}^T\mathbf{M}_1(P)\mathbf{v}(p) + \lambda\left[\frac{kc+1}{2}\mathbf{e}^T\mathbf{v}(t) + \frac{kc-1}{2}\ \mathbf{e}^T\mathbf{M}_2(P)\mathbf{v}(t) + sk\ \mathbf{e}^T\mathbf{M}_1(P)\mathbf{v}(t)\right]$$

and rearrange. □

Definition 7.23. Define $\mathbf{M}_3(A)$ to be a function of a three-dimensional bivector A,

$$\mathbf{M}_3(A) = \frac{kc+1}{2}\mathbf{I}_3 + \frac{kc-1}{2}\ \mathbf{M}_2(A) + sk\ \mathbf{M}_1(A)$$

where $\mathbf{I}_3$ is the 3×3 identity matrix.

Theorem 7.24. *The 3×4 matrix $\mathcal{H}$ may be found directly from a generator $B = \phi P + tn$ as*

$$\mathcal{H} = [c^2\mathbf{I}_3 - s^2\ \mathbf{M}_2(P) + 2sc\ \mathbf{M}_1(P)\ ;\ \mathbf{M}_3(P)\mathbf{v}(t)]$$

where $\mathbf{I}_3$ is the 3×3 identity matrix.

Proof. Clear by comparison of lemma 7.22 with definition 7.23. □

The required 4×4 matrix $\mathcal{R}$ can now easily be found from $\mathcal{H}$.

7.4.3 Mapping $\mathcal{H}$ to the corresponding generator

In this section we develop a method to reconstruct a generator given only the transformation matrix. Note that this method can only reconstruct a generator up to a rotation of 2π due to deficiencies in the matrix representation.

Definition 7.25. Define the two sub-matrices of $\mathcal{H}$, $\mathbf{R}$ and $\mathbf{t}$

$$\mathcal{H} = [\mathbf{R}\,;\,\mathbf{t}]$$

as

$$\mathbf{R} = c^2\mathbf{I}_3 - s^2\,\mathbf{M}_2(P) + 2sc\,\mathbf{M}_1(P) \tag{7.5}$$

and

$$\mathbf{t} = \mathbf{M}_3(P)\mathbf{v}(t). \tag{7.6}$$

Both $\mathbf{R}$ and $\mathbf{t}$ may easily be extracted from $\mathcal{R}$. Given the anti-symmetric and symmetric nature of $\mathbf{M}_1(P)$ and $\mathbf{M}_2(P)$ it is clear that

$$\mathbf{R} + \mathbf{R}^T = 2\,[c^2\mathbf{I}_3 - s^2\,\mathbf{M}_2(P)]$$
$$\mathbf{R} - \mathbf{R}^T = 4sc\,\mathbf{M}_1(P).$$

Definition 7.26. The function $\alpha(\mathbf{A}, \mathbf{B})$ estimates s^2 and c^2 and returns $\frac{s}{c}$ given the constraints

$$\mathbf{B} = c^2\mathbf{I}_3 - s^2\mathbf{A}$$

and

$$c^2 + s^2 = 1.$$

There are four constraints in two unknowns and hence the system is over-constrained – in practice one would use a linear-least-squares estimator.

If $sc \neq 0$ then we may recover P by extracting elements of $\mathbf{R} - \mathbf{R}^T$ and renormalising. If $sc = 0$ then either $s = 0$ or $c = 0$. If $s = 0$ it implies $\phi = 2n\pi$, $c = 1$ and therefore $\mathbf{R} + \mathbf{R}^T = 2\mathbf{I}_3$ and we are free to choose P as we wish. If

1: Extract the sub-matrices $\mathbf{R}$ and $\mathbf{t}$ from $\mathcal{R}$.
2: $\mathbf{K} := \mathbf{R} - \mathbf{R}^T$.
3: $\mathbf{L} := \mathbf{R} + \mathbf{R}^T$.
4: $k := \mathbf{K}_{12}^2 + \mathbf{K}_{13}^2 + \mathbf{K}_{23}^2$
5: **if** $k \neq 0$ **then**
6: $\quad P := \frac{1}{\sqrt{k}} \left(\mathbf{K}_{12}^2 e_{12} + \mathbf{K}_{23}^2 e_{23} - \mathbf{K}_{13}^2 e_{31} \right)$
7: $\quad \psi := \tan^{-1} \left[\alpha \left(\mathbf{M}_2(P), \frac{1}{2}\mathbf{L} \right) \right]$
8: **else**
9: $\quad$ **if** $L = 2\mathbf{I}_3$ **then**
10: $\quad\quad P := e_{12}$
11: $\quad\quad \psi := 0$
12: $\quad$ **else**
13: $\quad\quad P := \mathbf{M}_2^{-1} \left(\frac{1}{2}\mathbf{L} \right)$
14: $\quad\quad \psi := \frac{\pi}{2}$
15: $\quad$ **end if**
16: **end if**
17: $t := \mathbf{v}^{-1} \left([\mathbf{M}_3(P)]^{-1} \right)$
18: $B := \psi P + tn$

Figure 7.1: Reconstruction of a generator from a 4×4 transformation matrix.

$c = 0$ then $s = \pm 1$ and $\mathbf{R} + \mathbf{R}^T = -2\mathbf{M}_2(P)$. We can then extract P from $\mathbf{M}_2(P)$. The sign of s, in this case, is arbitrary.

Assuming we have estimates for s, k and c from above, we may reconstruct $\mathbf{M}_3(P)$ directly from $\mathbf{t}$ and hence recover

$$\mathbf{v}(t) = \mathbf{M}_3^{-1}(P)\mathbf{t}.$$

Finally, given s, c and k, an estimate for ϕ can be made. This algorithm is outlined explicitly in figure 7.1.

In practice we might wish to use LU decomposition or similar rather than computing the matrix inverse if we deal with spaces with higher dimension-

ality. Similarly one would calculate the inverse tangent in terms of s and c directly so as to ensure the result for the correct quadrant is returned.

7.5 Chapter summary

In this chapter we investigated a key feature of the conformal model – all rotors representing conformal transformations are formed by exponentiating bivectors. We found a closed form for the exponentiation of bivectors which results in rotors representing rigid body transforms and found that such bivectors lie, for 3d transformations, in a linear 6d subspace of the 5d bivectors. We also, importantly, found a method of converting back from a rotor into the corresponding bivector. Finally we used these relations to find algorithms for converting directly between 4×4 rotations/translation matrices and the 6d generators.

"I can accept anything, except what seems to be the easiest for most people: the half-way, the almost, the just-about, the in-between."

— Ayn Rand

8

Rotor Interpolation

8.1 Interpolation via Generators

We have shown that any displacement of Euclidean geometry may be mapped smoothly onto a linear subspace of the bivectors. This immediately suggests applications to smooth interpolation of displacements. Consider a set of poses we wish to interpolate, $\{P_1, P_2, ..., P_n\}$, and a set of rotors which transform some origin pose to these target poses, $\{R_1, R_2, ..., R_n\}$. We may map these rotors onto the set of bivectors $\{\ell(R_1), \ell(R_2), ..., \ell(R_n)\}$ which are simply points in some linear subspace of the bivectors. We may now choose any interpolation of these bivectors which lies in this space and for any bivector on the interpolant, B'_λ, we can compute a pose, $\exp(B'_\lambda)$. We believe this method is more elegant and conceptually simpler than many other approaches based on Lie algebras [24, 44, 48, 9].

Another interpolation scheme is to have the poses defined by a set of chained rotors so that $\{P_1, P_2, ..., P_n\}$ is represented by

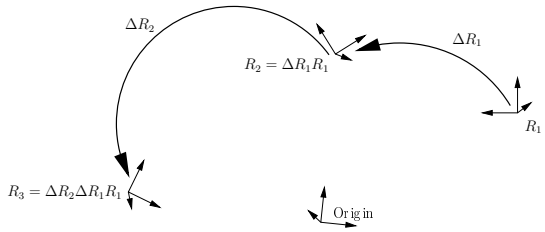


Figure 8.1: Rotors used to piece-wise linearly interpolate between key-rotors.

$$\{R_1, \Delta R_1 R_1, \Delta R_2 R_2, \ldots, \Delta R_n R_n\}$$

where $R_i = \Delta R_{i-1} R_{i-1}$ as in figure 8.1. Using this scheme the interpolation between pose R_i and R_{i+1} involves forming the rotor $R_{i,\lambda} = \exp(B_{i,\lambda}) R_{i-1}$ where $B_{i,\lambda} = \lambda \ell(\Delta R_{i-1})$ and λ varies between 0 and 1 giving $R_{i,0} = R_{i-1}$ and $R_{i,1} = R_i$.

We now investigate two interpolation schemes which interpolate through target poses, ensuring that each pose is passed through. This kind of interpolation is often required for key-frame animation techniques. The first form of interpolation is piece-wise linear interpolation of the relative rotors (the latter case above). The second is direct quadratic interpolation of the bivectors representing the final poses (the former case).

8.1.1 Piece-wise linear interpolation

Direct piece-wise linear interpolation of the set of bivectors is one of the simplest interpolation schemes we can consider. Consider the example shown in figure 8.1. Here there are three rotors to be interpolated. We firstly find

rotors, ΔR_n, which take us from rotor R_n to the next in the interpolation sequence, R_{n+1}.

$$\begin{aligned} R_{n+1} &= (\Delta R_n) R_n \\ \Delta R_n &= R_{n+1} \tilde{R}_n. \end{aligned}$$

We then find the bivector, ΔB_n which generates $\Delta R_n = \exp(\Delta B_n)$. Finally we form a rotor interpolating between R_n and R_{n+1}

$$R_{n,\lambda} = \exp(\lambda \Delta B_n) R_n$$

where λ is in the range $[0,1]$ and $R_{n,0} = R_n$ and $R_{n,1} = R_{n+1}$. Clearly this interpolation scheme changes abruptly at interpolation points, something which is reflected in the resulting interpolation as shown in figure 8.2a.

8.1.2 Quadratic interpolation

Another simple form for interpolation is the quadratic interpolation where a quadratic is fitted through three interpolation points, $\{B_1, B_2, B_3\}$ with an interpolation parameter varying in the range $(-1,+1)$

$$B'_\lambda = \left(\frac{B_3 + B_1}{2} - B_2 \right) \lambda^2 + \frac{B_3 - B_1}{2} \lambda + B_2$$

giving

$$B'_{-1} = B_1, \quad B'_0 = B_2 \text{ and } B'_{+1} = B_3$$

This interpolation varies smoothly through B_2 and is reflected in the final interpolation, as shown in figure 8.2b. Extensions to the quadratic interpolation for more than three interpolation points, such as smoothed quadratic interpolation [12], are readily available.

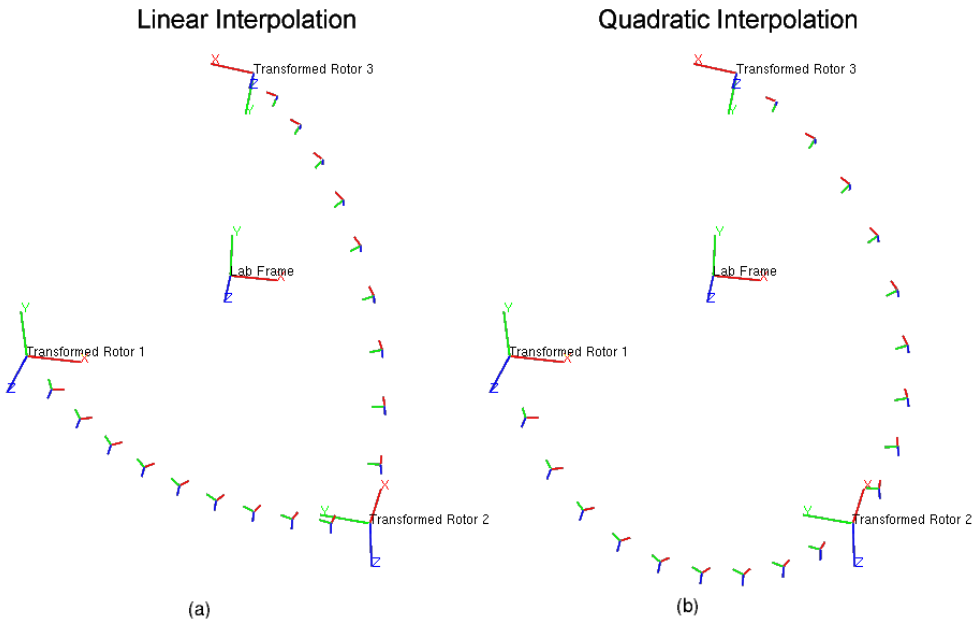


Figure 8.2: Examples of a) piece-wise linear and b) quadratic interpolation for three representative poses.

8.1.3 Alternate methods

It is worth noting that the methods described above used either a direct interpolation of the bivector $\ell(R)$ corresponding to a rotor R, as in the quadratic interpolation example, or by interpolating the relative rotors which take one frame to another, as in the piecewise linear example. Either method could have used either convention when choosing the bivectors to interpolate. Generally it is not clear which is the best approach and choosing that which fits a particular application may be the wisest course of action.

8.1.4 Interpolation of dilations

In certain circumstances it is desirable to add in the ability to interpolate dilations. This is investigated in [11] and is included here for completeness. In [11] it is shown that this can be done by extending the form of the bivector, B, which we exponentiate as follows

$$B = \phi P + tn + \omega N$$

where $N = e\bar{e}$. This bivector form is now sufficiently general [11] to be able to represent dilations as well. In this case obtaining the exponentiation and logarithm function is somewhat involved [11]. We obtain finally that

$$\begin{aligned}
&\exp(\phi P + tn + \omega N) \\
&\quad = \quad (\cos(\phi) + \sin(\phi)P)\,(\cosh(\omega) + \sinh(\omega)N + \mathrm{sinhc}(\omega)t_{J_\perp} n) \\
&\qquad + (\omega^2 + \phi^2)^{-1}[-\omega\sin(\phi)\cosh(\omega) + \phi\cos(\phi)\sinh(\omega)]P \\
&\qquad + (\omega^2 + \phi^2)^{-1}[\omega\cos(\phi)\sinh(\omega) + \phi\sin(\phi)\cosh(\omega)]t_{\|} n
\end{aligned}$$

where $\text{sinhc}(\omega) = \omega^{-1}\sinh(\omega)$. Note that this expression reduces to the original form for $\exp(B)$ when $\omega = 0$, as one would expect.

It is relatively easy to use the above expansion to derive a logarithm-like inverse function.

If we let $R = \exp(B)$ then we may recreate B from R using the method presented below. Here we use $\langle R|e_i\rangle$ to represent the component of R parallel to e_i, i.e. $\langle R|N\rangle = \langle R|e_{45}\rangle$ in 3-dimensions. We also use $\langle R\rangle_i$ to represent the i-th grade part of R and $S(X)$ to represent the 'spatial' portion of X (i.e. those components not parallel to e and $\bar{e}$).

$$
\begin{aligned}
\omega &= \tanh^{-1}\left(\frac{\langle R|N\rangle}{\langle R|1\rangle}\right) & W &= \langle R\rangle_2 - \cos(\phi)\sinh(\omega)N - \sin(\phi)\cosh(\omega)P \\
\phi &= \cos^{-1}\left(\frac{\langle R|N\rangle}{\sinh(\omega)}\right) & & -\cos(\phi)\text{sinhc}(\omega)t_{J_\perp}n \\
P &= \frac{S(\langle R\rangle_2)}{\sin(\phi)\cosh(\omega)} & X &= -\omega\sin(\phi)\cosh(\omega) + \phi\cos(\phi)\sinh(\omega) \\
t_\perp &= -\frac{\langle R\rangle_4 - \sin(\phi)\sinh(\omega)PN}{\sin(\phi)\text{sinhc}(\omega)}\left(\frac{P\bar{n}}{2}\right) & Z &= \omega\cos(\phi)\sinh(\omega) + \phi\sin(\phi)\cosh(\omega) \\
t &= t_{J_\parallel} + t_\perp & t_{J_\parallel} &= \frac{(-XP+Z)}{\sin^2(\phi)\cosh^2(\omega)+\cos^2(\phi)\sinh^2(\omega)}W
\end{aligned}
$$

8.2 Form of the Interpolation

In this section we derive a clearer picture of the precise form of a simple linear interpolation between two frames in order to relate the interpolation to existing methods used in mechanics and robotics. We will consider the method used above whereby the rotor being interpolated takes one pose to another.

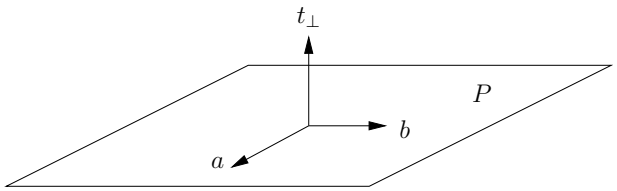


Figure 8.3: Orthonormal basis resolved relative to P.

8.2.1 Path of the linear interpolation

Since we have shown that $\exp(B)$ is indeed a rotor, it follows that any Euclidean pure-translation rotor will commute with it. Thus we only need consider the interpolant path when interpolating from the origin to some other point, since any other interpolation can be obtained by simply translating the origin to the start point. This location-independence of the interpolation is a desirable property in itself, but also provides a powerful analysis mechanism.

We have identified in section 7.1 the action of the $\exp(B)$ rotor in terms of $\phi, P, t_\parallel$ and $t_\perp$. We now investigate the resulting interpolant path when interpolating from the origin. We shall consider the interpolant $R_\lambda = \exp(\lambda B)$ where λ is the interpolation co-ordinate and varies from 0 to 1. For any values of $\phi, P, t_\parallel$ and $t_\perp$,

$$\lambda B = \frac{\lambda\phi}{2} P + \frac{\lambda(t_\perp + t_\parallel)n}{2}$$

from which we see that the action of $\exp(\lambda B)$ is a translation along $\lambda t_\perp$, a rotation by $\lambda\phi$ in the plane of P and finally a translation along

$$t'_\parallel = -\operatorname{sinc}\left(\frac{\lambda\phi}{2}\right)\lambda t_\parallel \left(\cos\left(\frac{\lambda\phi}{2}\right) - \sin\left(\frac{\lambda\phi}{2}\right) P\right).$$

We firstly resolve a three dimensional orthonormal basis relative to P as shown in figure 8.3. Here a and b are orthonormal vectors in the plane of P and hence $P = ab$. We may now express $t_{\|}$ as $t_{\|} = t^a a + t^b b$ where $t^{\{a,b\}}$ are suitably valued scalars.

The initial action of $\exp(B)$ upon a frame centred at the origin is therefore to translate it to $\lambda t_{\perp}$ followed by a rotation in the plane of P. Due to our choice of starting point, this has no effect on the frame's location (but will have an effect on the pose, see the next section).

Finally there is a translation along $t'_{\|}$ which, using $c = \cos\left(\frac{\lambda\phi}{2}\right)$ and $s = \sin\left(\frac{\lambda\phi}{2}\right)$, can be expressed in terms of a and b as

$$\begin{aligned} t'_{\|} &= -\frac{2s}{\lambda\phi}\lambda(t^a a + t^b b)(c - sab) \\ &= -\frac{2s}{\phi}\left[c(t^a a + t^b b) + s(t^b a - t^a b)\right] \\ &\equiv -\frac{2s}{\phi}\left[a(t^a c + t^b s) + b(t^b c - t^a s)\right]. \end{aligned}$$

The position, r_λ, of the frame at λ along the interpolation is therefore

$$r_\lambda = -\frac{2s}{\phi}[a(t^a c + t^b s) + b(t^b c - t^a s)] + \lambda t_{\perp}$$

which can easily be transformed via the harmonic addition theorem to

$$r_\lambda = -\frac{2s}{\phi}\alpha\left[a\cos\left(\frac{\lambda\phi}{2} + \beta_1\right) + b\cos\left(\frac{\lambda\phi}{2} + \beta_2\right)\right] + \lambda t_{\perp}$$

where $\alpha^2 = (t^a)^2 + (t^b)^2$, $\tan\beta_1 = -\frac{t^b}{t^a}$ and $\tan\beta_2 = -\frac{-t^a}{t^b}$. It is easy, via geometric construction or otherwise, to verify that this implies that $\beta_2 = \beta_1 + \frac{\pi}{2}$. Hence $\cos(\theta + \beta_2) = -\sin(\theta + \beta_1)$. We can now express the frame's position as

$$r_\lambda = -\frac{2\alpha}{\phi}\left[a\sin\left(\frac{\lambda\phi}{2}\right)\cos\left(\frac{\lambda\phi}{2} + \beta_1\right) - b\sin\left(\frac{\lambda\phi}{2}\right)\sin\left(\frac{\lambda\phi}{2} + \beta_1\right)\right] + \lambda t_{\perp}$$

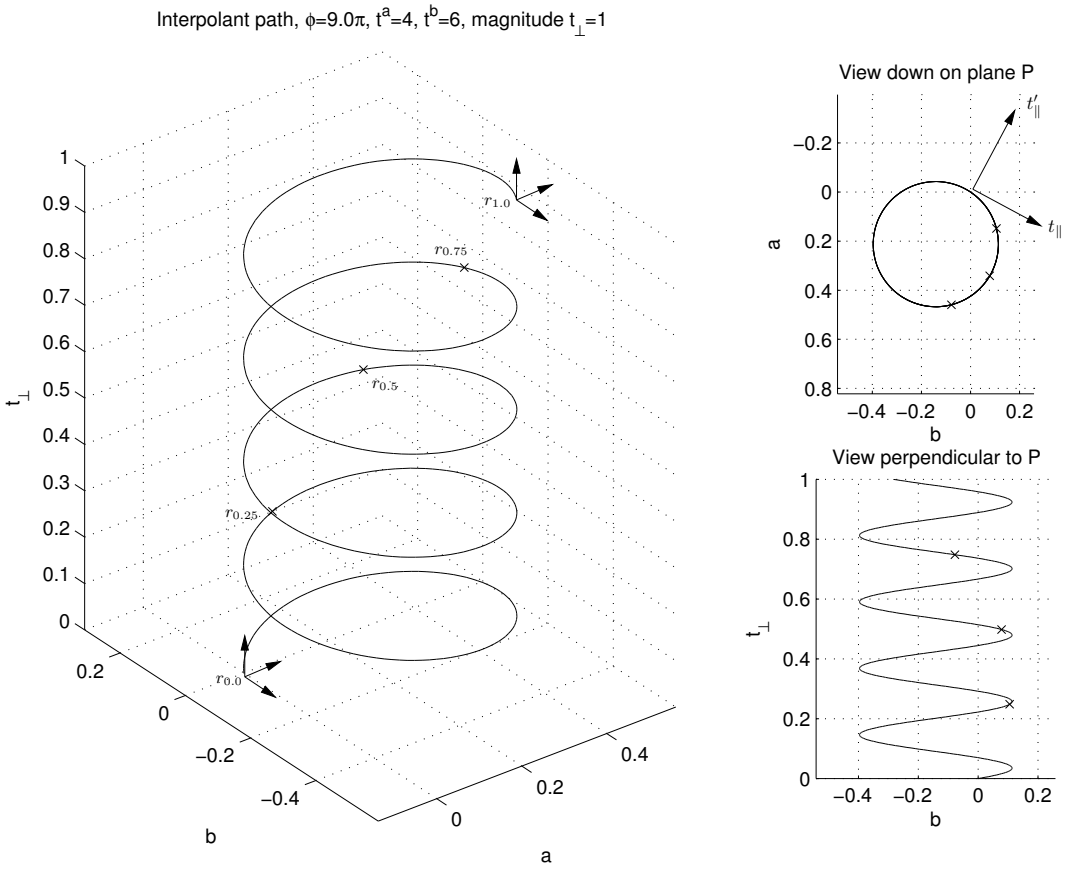


Figure 8.4: Example of an interpolant path with the final location being given by $t_{\parallel} = 4a + 6b$, $\phi = 9\pi$ and $t_{\perp}$ having a magnitude of 1.

which can be re-arranged to give

$$\begin{aligned} r_\lambda &= -\frac{\alpha}{\phi}\left[a\left(\sin\left(\lambda\phi+\beta_1\right)-\sin\beta_1\right)+b\left(\cos\left(\lambda\phi+\beta_1\right)-\cos\beta_1\right)\right]+\lambda t_\perp \\ &= -\frac{\alpha}{\phi}\left[a\sin\left(\lambda\phi+\beta_1\right)+b\cos\left(\lambda\phi+\beta_1\right)\right]+\frac{\alpha}{\phi}\left[a\sin\beta_1+b\cos\beta_1\right]+\lambda t_\perp \end{aligned}$$

noting that in the case of $\phi \to 0$, the expression becomes $r_\lambda = \lambda t_\perp$ as one would expect. Since a and b are defined to be orthonormal, the path is clearly some cylindrical helix with the axis of rotation passing through $\alpha/\phi\,[a\sin\beta_1 + b\cos\beta_1]$. An illustrative example, with a and b having unit magnitude, is shown in figure 8.4. It also clearly shows the relation between the direction of vector $t_\parallel$ and the final translation within the plane of P, $t'_\parallel$.

It is worth noting a related result in screw theory, Chasles' theorem, which states that any general displacement may be represented using a screw motion (cylindrical helix) such as we have derived. Screw theory is widely used in mechanics and robotics and the fact that the naïve linear interpolation generated by this method is indeed a screw motion suggests that applications of this interpolation method may be wide-ranging, especially since this method allows many other forms of interpolation, such as Bézier curves or three-point quadratic to be performed with equal ease. Also the pure rotation interpolation given by this method reduces exactly to the quaternionic or Lie group interpolation result allowing this method to easily extend existing ones based upon these interpolations.

8.2.2 Pose of the linear interpolation

The pose of the transformed frame is unaffected by pure translation and hence the initial translation by $\lambda t_{\perp}$ has no effect. The rotation by $\lambda\phi$ in the plane, however, now becomes important. The subsequent translation along $t'_{\parallel}$ also has no effect on the pose. We find, therefore, that the pose change λ along the interpolant is just the rotation rotor $R_{\lambda\phi,P}$.

8.3 Chapter summary

In this chapter we used the bivector to rotor mapping developed in the previous chapter to outline how existing interpolation schemes may be naturally extended to rotations and translations. Such interpolations tend to retain desirable properties when mapped from generators to rotors and allow for a far greater number of interpolation techniques to be applied to rotations and translations together than are currently used in computer graphics and animation. In subsequent chapters we shall see examples of computer graphic techniques which build upon this method of interpolation.

"They can't keep this level of graphics up for much longer! We used to be lucky if we only got three shades of grey, let alone any real colours!"

— Cranky Kong

9

Hardware Assisted Geometric Algebra on the GPU

In this chapter we explore how the *Graphics Processing Units* (GPUs) in modern consumer-level graphics cards can be used to perform Geometric Algebra computations faster than a typical single-core CPU.

Within the chapter a GPU implementation of certain GA algorithms is shown. It is important to note that these implementations are not designed to be a general-purpose implementation of GA such as Gaigen or CLUCalc. Instead these are special purpose routines to implement individual algorithms. This is because of the space constraints on GPU programs, a half-page GPU program is considered large, and so all solutions have to be domain specific to a degree.

In the future as general purpose computing on GPUs becomes more commonplace the space requirements may relax sufficiently to allow a general library to be written but until then we must work within the metaphorical tiny box given to us.

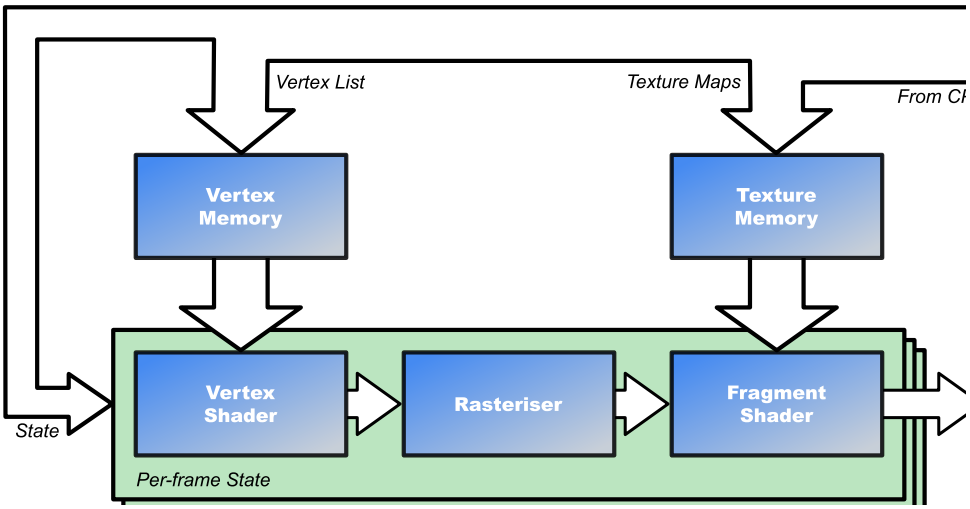


Figure 9.1: A simplified block diagram of a typical GPU.

9.1 An Overview of GPU Architecture

The GPU in a graphics card is not designed for general purpose computing. It is designed, unsurprisingly, to perform the sort of operations useful for graphics rendering. Figure 9.1 shows a simplified block diagram of a typical GPU.

In a traditional fixed-function GPU the CPU uploads, over the AGP bus, a set of vertices, texture maps and some state information. The state includes projection and view matrices, clipping planes, lighting models, etc. The vertex list and texture maps (if present) are then stored in some on-card memory.

On the card there exists a number of rendering pipelines, each of which can be run in parallel to increase throughput. The pipeline consists of a *vertex shader* which fetches triplets of vertices from the vertex memory, transforms them using the current projection and view matrices, performs any clipping

and sends them to the rasteriser.

The rasteriser takes triplets of screen co-ordinate vertices, forms a triangle from them and outputs a set of pixel positions from which the triangle is rendered along with depth information and interpolated texture co-ordinates.

The fragment shader takes these pixel positions and, using the texture maps stored in texture memory along with appropriate state information, calculates the colour of the pixel after all lighting, etc. is performed. The shaded pixel is then sent to the screen.

In reality the rendering stage is split at the rasteriser allowing for differing numbers of vertex and fragment shaders but for the purposes of GPU programming one can view the shaders as being within the same pipeline.

In modern programmable GPUs both the vertex and fragment shaders are fully programmable allowing different per-vertex and per-pixel transformation and shading that is allowed in the traditional fixed-function pipeline.

Each shader is, in effect, an efficient vector processor and, on modern graphics cards, there are a number working in parallel. To perform general purpose computation on the GPU we must find ways of modifying our algorithms to fit this model.

9.2 GPU Programming Methods

9.2.1 DirectX shader language

Microsoft's DirectX[42] is a graphics programming API for Windows and, in a modified form, the Microsoft XBox and XBox 360 gaming platforms. As

part of the API it specifies a generic shading language[43] which abstracts the vertex and fragment shaders. Each version of DirectX specifies a minimum set of shader capabilities which *must* be supported by a card claiming compatibility with that level of the API. Consequently a shader written in DirectX's shading language is portable across all cards which support that level of the API. A disadvantage of DirectX is that it does not expose any functionality beyond the specified minimum and it is not portable across operating systems.

9.2.2 OpenGL shader language

OpenGL[60] is a cross-platform C-based graphics API. It is the *de facto* standard API for non-Windows platforms and is well supported on Windows platforms by both ATI and nVidia, the dominant vendors at the time of writing.

The OpenGL 2.0 specification[58] proposed by 3DLabs includes a hardware-agnostic shading language[32] known as *GL Shading Language* (GLSL) which provides a similar level of functionality to that exposed in DirectX.

Currently few hardware vendors support OpenGL 2.0 — at the time of writing nVidia was the only mainstream vendor to provide support in their consumer-level hardware[47].

OpenGL 1.4 and below provide support for programmable shaders via a set of per-vendor extensions. Unfortunately these require programming the GPU in a variety of assembly languages and vary between different GPUs.

9.2.3 The Cg toolkit from nVidia

Both the OpenGL and DirectX shading languages have a number of disadvantages from the point of view of research. The OpenGL shading language, whilst being high-level and convenient to program in, is not widely available. The DirectX toolkit provides only a 'lowest common denominator' shading language, it is not particularly high-level and is, for all intents and purposes, limited to the Microsoft Windows operating systems.

An attempt to bridge these gaps is the Cg toolkit[45] from nVidia. It provides a high-level C-like shading language which may be compiled at run-time or ahead of time into either DirectX shading language or the myriad OpenGL extensions which exist for accessing the programmable shaders.

Along with these advantages is the 'future-proof' nature of Cg. The core Cg language is easily extended via so-called 'profiles'. These either restrict the language to correspond to the capabilities of particular GPUs or add more features to expose new GPU abilities. As an example in figure 9.1 it was implied that the vertex shader cannot access the texture-map memory. This was true of older cards but the latest generation of nVidia cards can access texture maps from within the vertex shader[46]. Adding this capability to Cg was simply a case of releasing a new profile where texture-map related parts of the language were now available from within both shaders. Older shaders would still work however and the presence of this capability could be queried at run-time.

Because of these advantages it was decided to use Cg as the basis for the

GPU computing research. None of the techniques described below require it as they can all be re-structured to use alternate APIs.

9.3 A Cg Implementation of Generator Exponentiation

In order to integrate the GA interpolation scheme described in the previous chapter into the GPU graphics pipeline it had to be re-cast into the mathematical language of the GPU. Graphics cards are optimised for linear algebra on 4-dimensional vectors and 4×4 matrices, reflecting the dominant language used in graphics.

It was decided to abstract away the GA method and represent rotors and generators as sets of vectors. The Cg interface and corresponding CPU-side C interface are shown in figure 9.2. The required GA operations for rotor exponentiation and application ($X \mapsto RX\tilde{R}$) were expanded out in terms of basis components and implemented directly in Cg. The routines could now be used as a 'black box' by the GPU programmer. Indeed no GA knowledge is required for their use, merely that one applies rotors to points to transform them, one can convert generators into rotors and generators may be linearly interpolated.

A more general GA-solution, such as `libcga`, was impractical given the current space constraints on Cg programs although future GPUs may provide enough space to make such a library feasible.

```
/* Cg interface for generator exponentiation and rotor operations */

/* Rotors are represented as an array of two four-dimensional vectors to form
 * an eight-dimensional vector which represents the following components of
 * the rotor:
 *
 *    [ e_0, e_12, e_23, e_31, e_14, e_24, e_34, e_1234 ]
 *
 * where e = e_4 and \bar{e} = e_5. We need not represent the components with
 * parts parallel to e_5 since we may trivially reconstruct them for
 * rigid-body rotors.
 */

/* Generators are represented as na array of two three-dimensional vectors[1]
 * to form a six-dimensional vector which represents the following components
 * of the generator:
 *
 *    [ e_12, e_23, e_31, e_14, e_24, e_34 ]
 *
 * where e = e_4 and \bar{e} = e_5. Again the components containing e_5 can be
 * reconstructed.
 *
 * [1] When compiled for the GPU this may well be expanded out to two
 * /four/-dimensional vectors but it is convenient to split the generator such
 * in the implementation.
 */

/* Sets rotor[] to contain the exponentiation of generator[] */
void exp_generator(in float3 generator[2], out float4 rotor[2]) {
  ...
}

/* Returns the point the origin is mapped to by rotor[] */
float3 apply_rotor_to_origin(in float4 rotor[2]) {
  ...
}

/* Returns the point x is mapped to by rotor[] */
float3 apply_rotor_to_point(in float4 rotor[2], in float3 x) {
  ...
}
```

```
/* C interface for generator exponentiation and rotor operations */

/* Like the Cg interface rotors are represented as eight-dimensional vectors
 * and generators by six-dimensional vectors. Unlike Cg there are no vector
 * primitive types in C and so appropriately dimensioned arrays of floats are
 * used. */

/* Sets rotor[] to contain the exponentiation of generator[] */
void exp_rotor(float generator[6], float rotor[8]);

/* Sets x to the point the origin is mapped to by rotor[] */
void apply_rotor_to_origin(float rotor[8], float x[3]);

/* Overwrites x with the point x is mapped to by rotor[] */
void apply_rotor(float rotor[8], float x[3]);
```

Figure 9.2: The Cg and C interfaces for dealing with rotors and exponentiated generators.

9.4 Mesh Deformation

In this section we shall give an example of a vertex shader that uses GA to perform mesh deformation. The deformation scheme given is a simple application of Geometric Algebra but nicely shows the applicability of GA to a wide variety of problems. The method developed has a number of nice properties, but it is intended as an illustration of the power of GA algorithms when implemented on the GPU. It may be useful in its own right if researched further but the field of mesh deformation is large and beyond the scope of this discussion.

9.4.1 Method

We shall now present a GA-based mesh deformation scheme suitable for implementation on a GPU. In this scheme we start with an existing mesh and set of key rotors, $\{R_1, R_2, \cdots, R_{N_k}\}$, which we wish to use to deform the mesh. Our scheme seeks to find some automatic method of representing each point on the mesh as some function of the key rotors; changing the key rotors will then lead to a deformation of the mesh. Desirable properties include smoothness, small changes in the rotors lead to small changes in the mesh, and to be intuitive, i.e. changes in the rotors should produce changes in the mesh which a user, ignorant of the method, would expect.

Our assignment scheme is illustrated in figure 9.3. Firstly we form a set of generators, $\{B_1, B_2, \cdots, B_{N_k}\}$, such that

$$\{R_1, R_2, \cdots, R_{N_k}\} \equiv \{\exp(B_1), \exp(B_2), \cdots, \exp(B_{N_k})\}.$$

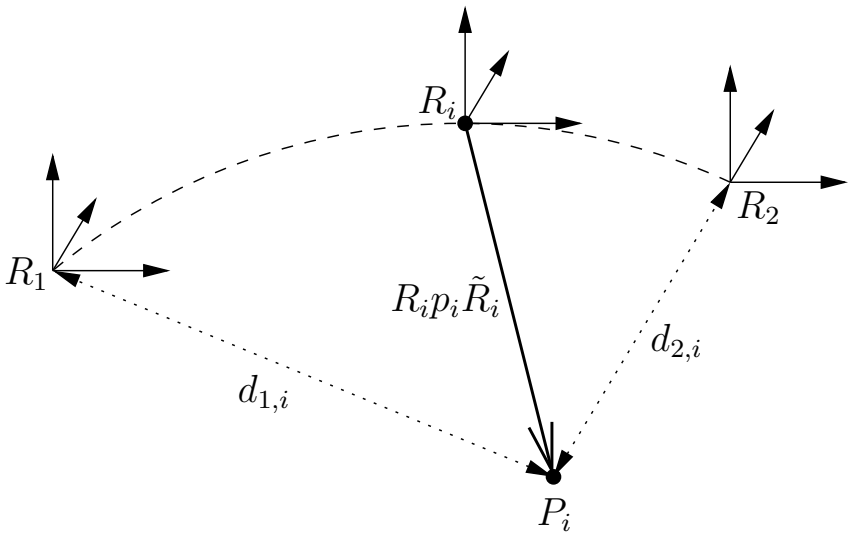


Figure 9.3: Representing a point, P_i, on a mesh as a rotor, R_i, and displacement, p_i, given a set of key rotors, $\{R_1, R_2\}$.

We also find the image of the origin, $O_k = R_k \bar{n} \tilde{R}_k$, for each key rotor. For each point-representation on the mesh, P_i, we find the Euclidean distance from it to the image of the origin in each key rotor,

$$d_{k,i} = \sqrt{-2 O_k \cdot P_i}.$$

We then form a rotor, R_i, for each point on the mesh as a weighted sum of the key rotors,

$$R_i = \exp\left(\frac{\sum_{k=1}^{N_k} w_i(d_{k,i}) B_i}{\sum_{k=1}^{N_k} w_i(d_{k,i})}\right)$$

where $w_i(d)$ is a function which defines the relative weights of each rotor depending on its proximity to the mesh point. One might choose a relatively simple weight function

$$w_i(d) = \frac{1}{d + \varepsilon}$$

where ε is a small value to avoid a singularity when the key rotor and mesh intersect. In our implementation we wished to have key rotors with varying degrees of influence. This was accomplished by using a Gaussian weight function,

$$w_i(d) = \exp\left(-\frac{d^2}{\sigma_i^2}\right)$$

with σ_i giving the radius of influence of a particular key rotor.

Finally we find a point, p_i, which is transformed by R_i to coincide with P_i,

$$(R_i p_i \tilde{R}_i) \cdot P_i = 0$$

which may be found by simply applying the reversal of R_i to P_i

$$p_i = \tilde{R}_i P_i R_i.$$

The point P_i is now stored in memory as a set of $d_{k,i}$ for each key rotor and the point p_i.

Our deformation procedure is now simple. Given a new set of key rotors, $\left\{R'_1, R'_2, \cdots, R'_{N_k}\right\}$, and a corresponding set of generators, $\left\{B'_1, B'_2, \cdots, B'_{N_k}\right\}$, we can form the deformed mesh point P'_i given the previously calculated $d_{k,i}$ and p_i as

$$R'_i = \exp\left(\frac{\sum_{k=1}^{N_k} w_i(d_{k,i}) B'_i}{\sum_{k=1}^{N_k} w_i(d_{k,i})}\right) \tag{9.1}$$

$$P'_i = R'_i p_i \tilde{R}'_i. \tag{9.2}$$

It is easy to show by direct substitution that for $R'_k = R_k$ this reduces to $P'_i = P_i$ as would be expected.

9.4.2 GPU-based implementation

Since the GPU distinguishes between global state and per-vertex information we must decide what information needs to be given to the GPU and how. In our mesh deformation example we need the key rotors which are part of the global state and, for each mesh point, the vector p_i and set of distances $d_{k,i}$.

In OpenGL each vertex has at least a three-dimensional position vector associated with it and may have a normal vector. In addition there are a number of texture co-ordinates which may be associated with each vertex. In our implementation the values of $w(d_{k,i})$ will be stored into the texture co-ordinates. The weight function is pre-computed to save time.

Figure 9.4 gives the vertex shader used to perform mesh deformation in this example. Line 2 includes the standard set of rotor manipulation functions which were described above.

The generators associated with the key rotors are passed in the state array `generators[][]` and are constant for a particular scene. Lines 32 to 39 perform the summation in equation 9.1 and line 41 uses the function `exp_generator()` to form R'_i. Line 42 applies R'_i to the point p_i and the remainder of the shader is boilerplate code to project into screen-space and perform a simple lighting calculation.

The algorithm and OpenGL code required to compute the rotor weights and offset-vector p_i for each mesh vertex is shown in algorithm 9.5. Note that this need only be performed once per vertex at initialisation.

In the actual implementation *display lists* were utilised. A display list is

```
1  /* Include Cg implementations of rotor exponentiation and 'logarithm' */
2  #include "rotor_tools.cg"
3
4  /* Per-vertex input */
5  struct appin {
6    float4 Position  : POSITION;  /* Store p_i as the vertex position */
7    float4 Normal    : NORMAL;    /* Normal vector */
8    float4 Diffuse   : DIFFUSE;   /* Diffuse colour */
9
10   /* Store d_{k,i} in the texture co-ordinates */
11   float2 Coeffs1 : TEXCOORD0; float2 Coeffs2 : TEXCOORD1;
12   float2 Coeffs3 : TEXCOORD2; float2 Coeffs4 : TEXCOORD3;
13 };
14
15 /* Per-vertex output */
16 struct vertout {
17   float4 HPosition : POSITION;  /* Screen-space position */
18   float4 Color     : COLOR;     /* Colour after lighting */
19 };
20
21 vertout main(appin IN, const uniform float4x4 ModelViewProj : _GL_MVP,
22                        uniform float4x4 ModelViewIT, uniform float4x4 ModelView,
23                        uniform float3 generators[8][2])
24 {
25   vertout OUT; float4 p = IN.Position;
26   float3 generator[2]; float4 rotor[2]; /* For calculating R'_i */
27
28   /* Initialse generator to be zero */
29   generator[0] = float3(0,0,0); generator[1] = float3(0,0,0);
30
31   /* Perform summation */
32   generator[0] += generators[0][0] * IN.Coeffs1.x; generator[1] += generators[0][1] * IN.Coeffs1.x;
33   generator[0] += generators[1][0] * IN.Coeffs1.y; generator[1] += generators[1][1] * IN.Coeffs1.y;
34   generator[0] += generators[2][0] * IN.Coeffs2.x; generator[1] += generators[2][1] * IN.Coeffs2.x;
35   generator[0] += generators[3][0] * IN.Coeffs2.y; generator[1] += generators[3][1] * IN.Coeffs2.y;
36   generator[0] += generators[4][0] * IN.Coeffs3.x; generator[1] += generators[4][1] * IN.Coeffs3.x;
37   generator[0] += generators[5][0] * IN.Coeffs3.y; generator[1] += generators[5][1] * IN.Coeffs3.y;
38   generator[0] += generators[6][0] * IN.Coeffs4.x; generator[1] += generators[6][1] * IN.Coeffs4.x;
39   generator[0] += generators[7][0] * IN.Coeffs4.y; generator[1] += generators[7][1] * IN.Coeffs4.y;
40
41   exp_generator(generator, rotor);              /* Exponentiate to form R'_i ... */
42   p.xyz = apply_rotor_to_point(rotor, p);       /* ... and apply it to p_i to get our final mesh point */
43
44   /* Set translation part of the rotor to zero and rotate the normal to match the new mesh point. */
45   rotor[1] = float4(0,0,0,0); float4 normalVec = mul(ModelViewIT, IN.Normal);
46   normalVec.xyz = normalize(apply_rotor_to_point(rotor, normalVec.xyz));
47
48   /* Project mesh point to screen-space and light it as usual. */
49   OUT.HPosition = mul(ModelViewProj, p);
50   float3 lightVec = normalize(float3(0,0,-5));
51   OUT.Color = (0.3 + 0.7*lit(dot(normalVec.xyz, lightVec.xyz),0,0).y ) * IN.Diffuse;
52   OUT.Color.a = IN.Diffuse.a;
53
54   return OUT;
55 }
```

Figure 9.4: The vertex shader used to perform GA-based mesh deformation.

Require: P_i, the null-vector representation of mesh vertex i.
Require: $\mathcal{B} \equiv \{B_1, B_2, \cdots, B_k, \cdots\}$, the set of key rotor generators.

1: $w_{\text{sum}} := 0$
2: **for** $k := 1$ to $n(\mathcal{B})$ **do**
3: $\quad R := \exp(B_k)$
4: $\quad O := R\bar{n}\tilde{R}$
5: $\quad w_k := w(\sqrt{-2O \cdot P})$
6: $\quad w_{\text{sum}} := w_{\text{sum}} + w_k$
7: **end for**
8: **for** $k := 1$ to $n(\mathcal{B})$ **do**
9: $\quad w_k := \frac{w_k}{w_{\text{sum}}}$
10: **end for**
11: $p := \tilde{R} P_i R$
12: glMultiTexCoord2f(GL_TEXTURE0, w_1, w_2);
13: glMultiTexCoord2f(GL_TEXTURE1, w_3, w_4);
14: glMultiTexCoord2f(GL_TEXTURE2, w_5, w_6);
15: glMultiTexCoord2f(GL_TEXTURE3, w_7, w_8);
16: glVertex(p);

Figure 9.5: Algorithm for computing $w(d_{k,i})$ and p_i for each mesh point and storing them in the texture co-ordinates and vertex position. In this case there are eight key rotors.

an OpenGL technique for uploading a set of OpenGL calls to the graphics card and executing them again with one call. Since the vertices and texture co-ordinates generated by algorithm 9.5 don't change once calculated each mesh could be processed once and sent to the graphics card as a display list.

Using a display list meant that the deformation now became extremely efficient in terms of CPU usage. For each frame all that needed to be done by the CPU was update the state variables holding the key rotor generators and ask for the display list to be drawn. Using such a technique resulted

in the Unix *top* utility reporting 0% CPU utilisation, i.e. below measurable resolution.

9.4.3 Quality of the deformation

The quality of a mesh deformation technique is ultimately subjective. A simple example is shown in figure 9.6. In this example there were eight key rotors labelled '000' to '111' in binary. The rotors were moved to positions within a rabbit model and appropriate weighting and offset-vectors were assigned using algorithm 9.5. The '001' rotor, which was positioned within the rabbit's head, was then moved and the results are shown. The movement is smooth, natural and intuitive.

Figure 9.7 shows the natural 'plasticine-like' effect the deformation scheme has on a unit cube with key rotors initially placed on its corners. In addition to this figure 9.8 shows the effect of placing the key rotors along a central axis and applying opposite rotations at either end. Notice how the cube behaves as expected and does not collapse in the middle.

9.4.4 Performance

To test the relative performance of software and hardware two implementations were made, one software and one hardware. Both implemented the same mesh deformation algorithm as above and both used as near equal, within the intersection of C and Cg, implementations of the generator exponentiation and rotor application routines.

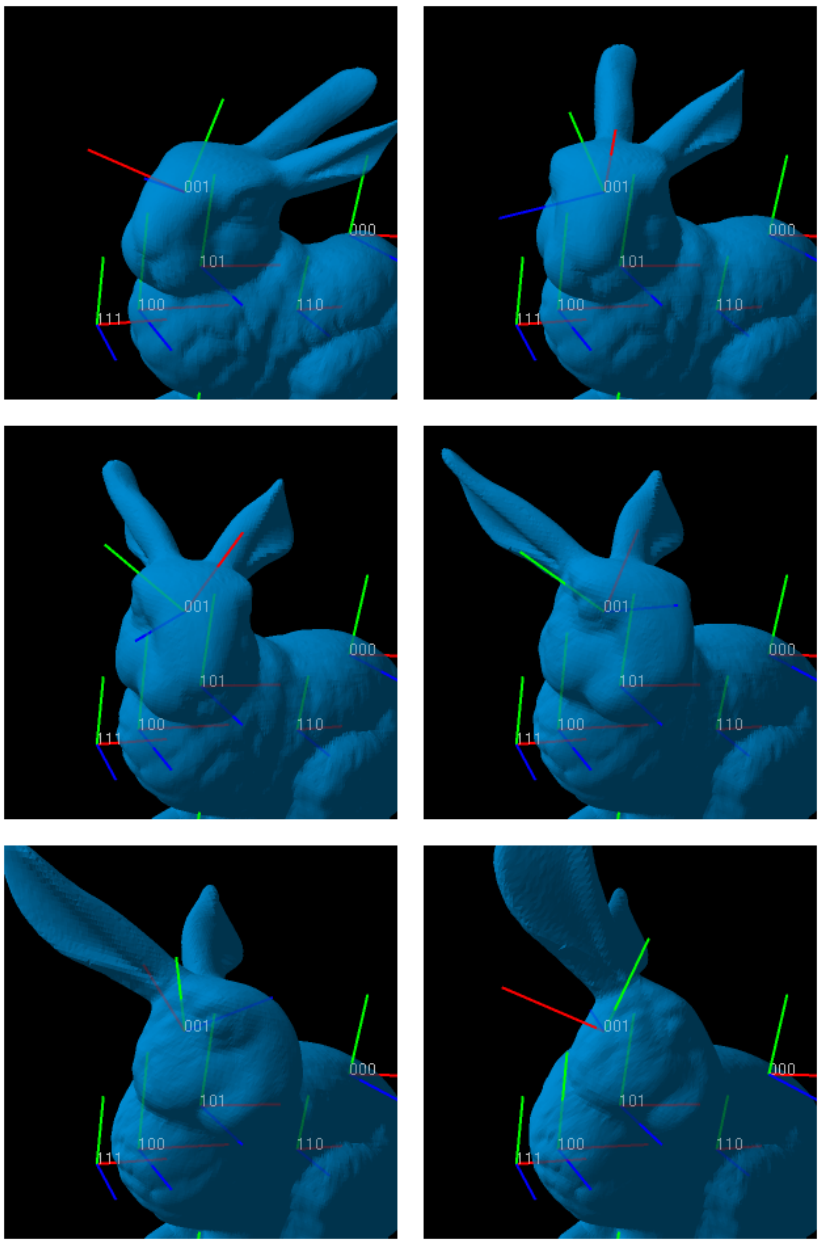
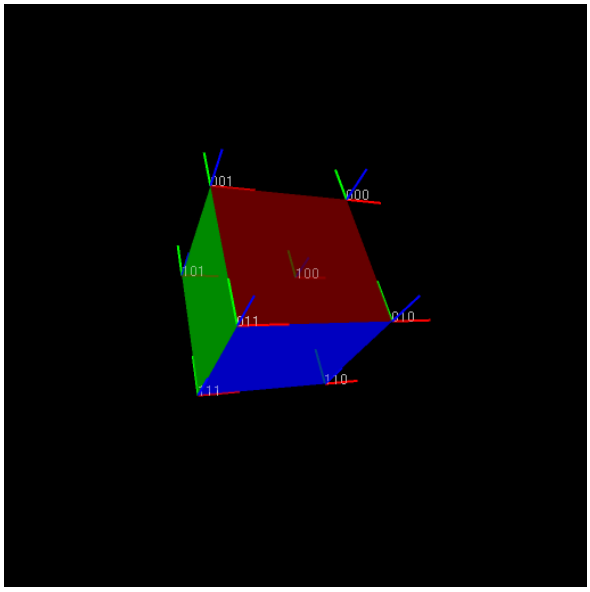
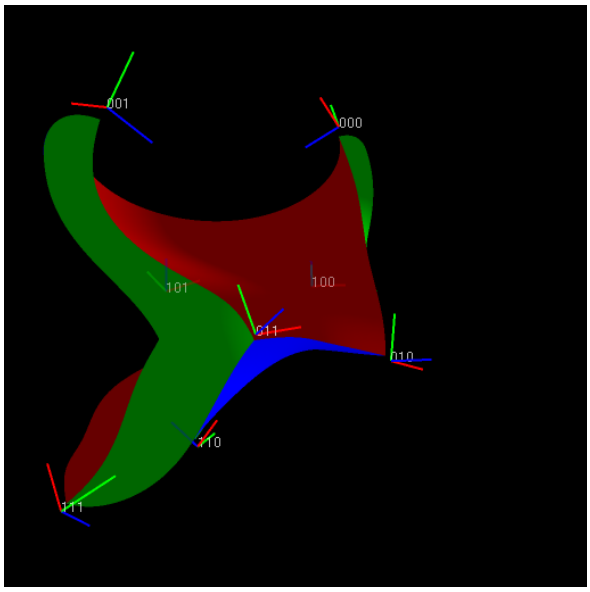


Figure 9.6: An example of animating a rabbit's head using key rotors and an automatically assigned mesh.

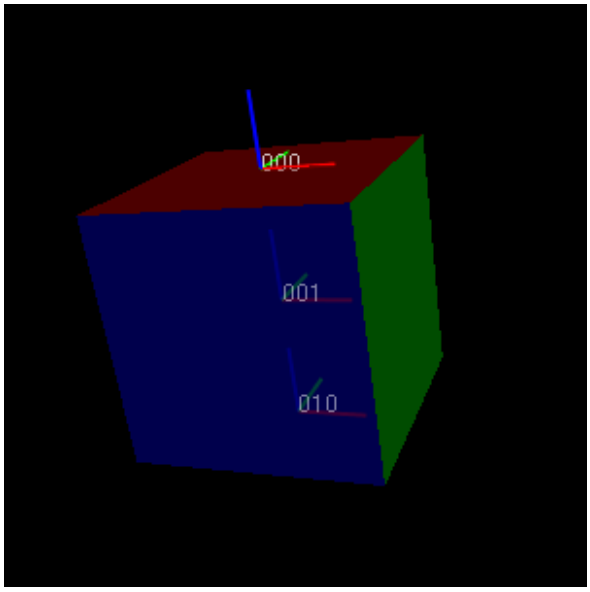


(a)

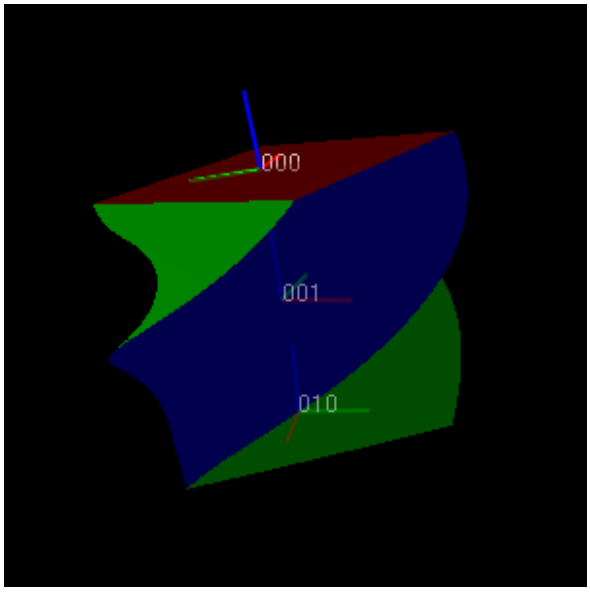


(b)

Figure 9.7: An example of mesh deformation acting on a unit cube. (a) Initial key rotors and automatically assigned mesh. (b) Deformed mesh after movement of key rotors.



(a)



(b)

Figure 9.8: An example of screw deformation acting on a unit cube. (a) Initial key rotors and automatically assigned mesh. (b) Twisted mesh after movement of key rotors.

	Frames per second		
Polygons	Hardware	Software	Ratio
14,406	219.10	18.81	11.65:1
20,886	160.80	12.50	12.86:1
29,400	119.50	9.08	13.16:1
43,350	83.91	6.26	13.40:1
60,000	62.13	4.65	13.36:1

	Frames per second		
Polygons	Hardware	Software	Ratio
93,750	20.42	2.89	7.07:1
135,000	14.20	2.07	6.86:1
240,000	8.13	1.14	7.13:1
372,006	5.32	0.75	7.09:1
–	–	–	–

Table 9.1: The relative performance, in frames per second, between the GPU and pure-software mesh deformation implementations.

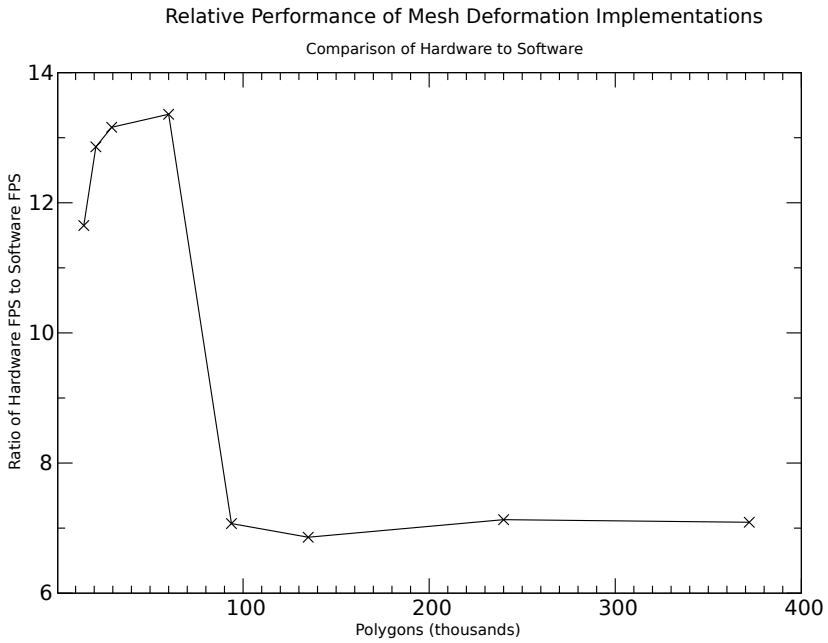


Figure 9.9: A plot of the ratio between FPS using the GPU-based implementation and the pure-software implementation.

The implementations differed most in the use of display lists. To mirror real-world practices each model in the hardware implementation was uploaded to the graphics card in a display list, since the per-vertex set of $d_{k,i}$ and p_i could be pre-computed. In the software implementation these were also pre-computed, but the deformed vertices had to be uploaded to the graphics card once-per frame since the deformation step was done in software.

Table 9.1 shows the number of frames per second that could be displayed with a simple cube model at various different polygon counts. A simple model was chosen so that the generation, per frame, of un-deformed model vertices in the software implementation would take as little time as possible, being algorithmically generated rather than fetched from main memory providing a fairer test of the speeds of the deformation algorithm. Figure 9.9 shows the ratio of improvement between software and hardware implementations with polygon count.

It is interesting to note the sudden dip in performance around 100,000 polygons. Since the internals of GPUs are not publicly available it is only possible to speculate as to the reason of this but it might be related to vertex cache size. If all of the model vertices fit within the on-GPU vertex cache they may be processed efficiently without accessing graphics memory. If the number of vertices exceed the vertex cache size then they must be copied into the cache in batches which slows performance.

It is interesting to note the sudden dip in performance around 100,000 polygons. Since the internals of GPUs are not publicly available it is only

possible to speculate as to the reason of this but it might be related to vertex cache size. If all of the model vertices fit within the on-GPU vertex cache they may be processed efficiently without accessing graphics memory. If the number of vertices exceed the vertex cache size then they must be copied into the cache in batches which slows performance.

9.5 Dynamics

In this section we will develop a simple method for doing dynamics with a sphere which has been deformed with a set of rotors. We shall show how a simple dynamics example using such a method may be implemented on the GPU.

Recall that a GPU has two classes of shaders. It has vertex shaders which are applied per-vertex and fragment shaders which are applied per-pixel. Since, in a typical scene, one would expect the number of pixels on screen to be very much greater than the number of vertices, GPUs generally have more parallel fragment shaders than vertex shaders. If we can re-formulate our solution to use fragment shaders we might expect even greater performance than simply using the vertex shader.

Algorithms implemented on the fragment shaders have one further advantage when compared to those implemented on the vertex shader when one makes use of the *render to texture* feature on modern graphics cards. Using this feature, rendering can be directed to a texture stored in the graphics card memory rather than the screen. This feature allows iterative algorithms

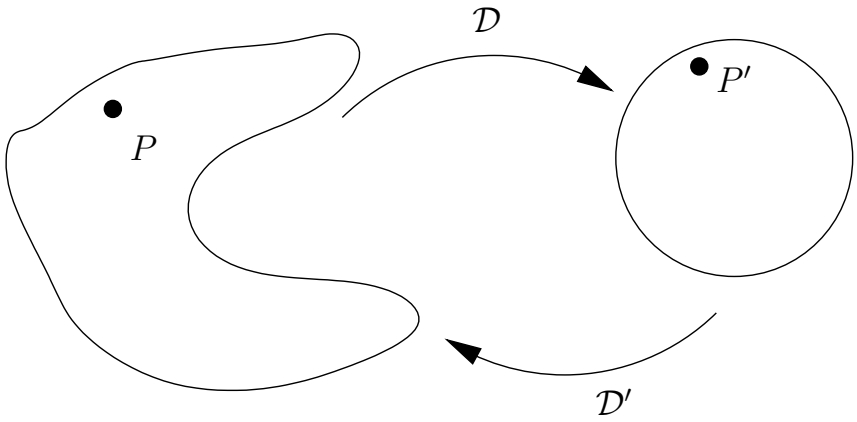


Figure 9.10: Given a deformation scheme $\mathcal{D}$ which maps our object to the unit sphere we can tell whether a point, P, is inside the object by testing if the mapped point, P', is inside the sphere.

to be developed. The concept is simple. A texture is created which stores a set of initial values. A square is then rendered with a fragment shader which, for each pixel in the square, reads the initial value from the texture and outputs the result of the next iteration. If this square is rendered into the original texture then the result of each iteration replaces the previous one. This process may be repeated as often as is desired. In reality there are a few implementation issues. Aside from the API calls required to set up the render to texture and appropriate projection matrices, a significant issue is that the shader is required to write back to its input which could lead to concurrency issues. To avoid this one generally uses two textures, an 'input' and an 'output', which are swapped between each iteration.

9.5.1 Collision detection via deformation

We shall develop a simple example to illustrate this method. In our example we shall implement an approximate simulation of a cloth falling onto a complex smooth object.

To begin with we need a GPU-based simulation of the cloth itself, for when it is moving in space away from the target object. The aim of this example is to demonstrate complex collision rather than cloth simulation *per se* and so we choose a very simple ball and spring model; the cloth is composed of a $N \times M$ grid of masses connected by simple Hookian springs. Formally, if we let $P_{i,j}$ be the position vector of the ball in row i, column j then the force acting on it, $F_{i,j}$ is

$$F_{i,j} = \sum_{\alpha \in \{-1,1\}} \sum_{\beta \in \{-1,1\}} b_{i+\alpha,j+\beta}(P_{i,j} - P_{i+\alpha,j+\beta})$$

where

$$b_{i,j} = \begin{cases} 1 & \text{if } i \in \{1,\cdots,N\}, j \in \{1,\cdots,M\} \\ 0 & \text{otherwise} \end{cases}$$

The variation of $P_{i,j}$ over time may then be obtained by numerically integrating the force twice.

Such simple dynamics are often employed in games where the appearance of correct physics is often more desirable, if it is faster, than a full 'correct' calculation. We shall use a common game technique which may be summarised as 'detect and backtrack'.

Suppose a cloth vertex were known to be outside of the target object at time t and we detect it is to be inside the object at time $t + \Delta t$. We stop the

simulation and backtrack to time $t + \Delta t - t_{\text{offset}}$ with $t_{\text{offset}} < \Delta t$ such that the vertex is just touching the target object. We then use simple surface physics to modify the total force acting on the vertex to cause reflection, friction or any other surface property we may wish. The simulation is then restarted from this point.

We shall discuss the precise implementation of the cloth simulation later but for the moment we note that the key operation is detecting the interpenetration of the grid of cloth vertices and the target object and being able to move penetrating vertices to the surface of the object.

Our approach is illustrated in figure 9.10. We shall assume some invertible, locally conformal, GA-based deformation scheme $\mathcal{D}$ which will deform the unit sphere to our target object. We now apply the inverse scheme $\mathcal{D}$ to both our target object and the set of cloth vertices. The target object is mapped to the unit sphere and the set vertices are mapped to a different set of vertices. Our penetration test for vertices is simply to see if its distance from the origin is less than unity. We can thereby identify all penetrating vertices. Any penetrating vertex can be corrected simply by moving the deformed vertex to the unit-sphere surface. If we re-apply the deformation scheme the unit sphere is mapped back to the target object and the set of corrected vertices is mapped to a set which all lie outside the target object.

9.5.2 A suitable deformation scheme

We wish our deformation scheme to be locally conformal and, preferably, for the normal information associated with any point on the target object to be preserved for lighting and dynamics. Below we present a simple scheme based upon weighted sums of pure rotation generators which fulfils this requirement.

We begin by considering a rotor, $R_i = \exp(B_i)$, representing a rotation around a known point P_i. Clearly $R_{i,\lambda} = \exp(\lambda B_i)$ is also a pure-rotation for some scalar λ. To deform a particular point representation, $X = F(x)$, using the key-generator B_i we perform the mapping

$$X \mapsto R_{i,w(X,P_i)} X \tilde{R}_{i,w(X,P_i)}$$

where $w(A,B)$ is a weighting function with range $[0,1]$. For a simple deformation scheme we might choose $w(\cdot)$ to be

$$w(A,B) = \begin{cases} 1 - d(A,B) & \text{if } d(A,B) < 1 \\ 0 & \text{otherwise} \end{cases}$$

where $d(A,B)$ is the Euclidean distance metric defined in equation 2.8. This is of course a linear fall-off. One could easily create non-linear exponential or sigmoid fall-off which would lead to smoother deformation at the cost of greater computational complexity.

The effect of the deformation scheme is shown diagrammatically in figure 9.11b. Here four unit lengths are deformed by rotating points on a set of spherical shells centred on P_i. On each spherical shell the deformation is

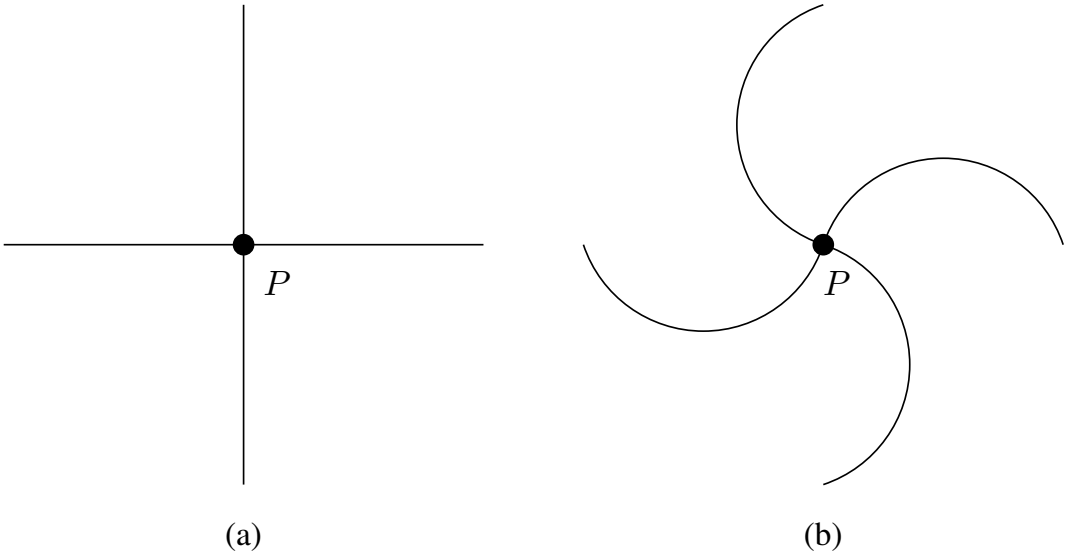


Figure 9.11: Diagram illustrating weighted generator deformation around a point P. a) Undeformed state. b) Effect of weighted rotation deformation.

therefore locally conformal. Normal to the shells the deformation is less so but, as is often the case in Computer Graphics, the errors introduced by this may be ignored since they are perceptually slight.

Since we are using position-dependent rotors we can view the deformation not as a point-wise deformation scheme but a full local-frame deformation scheme. For example we can represent an orientation associated with X as a pure-rotation rotor, R_X, which is applied to some known reference orientation. The post-deformation orientation is now

$$R_X \mapsto R_{i,w(X,P_i)} R_X .$$

Specifically, if the point X is a mesh vertex with associated normal $\hat{n} = R_X e_1 \tilde{R}_X$

then we can find the deformed normal, $\hat{n}'$, as

$$\hat{n}' = R_{i,w(X,P_i)} R_X e_1 \tilde{R}_X \tilde{R}_{i,w(X,P_i)}.$$

This 'frame deformation' property is a key one when it comes to performing perceptually valid dynamics on deformed objects.

9.5.3 Implementation

There are two stages to each step of the simulation, firstly the location of the cloth vertices are updated followed by the detection and correction of vertices which penetrate the object. For these vertices we perform some basic surface physics removing force components normal to the surface and crudely modelling friction by multiplying the tangential velocity component by some fixed friction coefficient between 0 and 1.

Both of these steps are performed on the GPU. Unlike the previous example this simulation required feedback. The output from the simulation had to be fed back to the input.

The first stage in the simulation is the calculation of forces, velocities and positions for each cloth vertex.

The location of each cloth vertex was stored in a texture with each pixel in the texture corresponding to a vertex in the cloth. The red, green and blue components specifies the x, y and z co-ordinates respectively. Many GPUs only allow for texture components to be in the range $[0, 1]$, effectively restricting the cloth to the unit cube. This could have been overcome with the introduction of appropriate scaling constants, but newer nVidia GPUs allow for

float buffers which remove this restriction and they were used in this implementation due to their increased accuracy and convenience. A simple vertex shader could then be used when rendering the cloth to position the vertices according to the contents of the texture. In addition to the vertex texture a separate texture was created holding the velocity of each cloth vertex in a similar manner.

To update these textures we made use of *render to texture* support in newer GPUs where pixel and vertex shaders can be used to render images, not to screen, but to a particular texture. By rendering a single quad (or two triangles) into a texture we could invoke the pixel shader for each texel.

GPUs cannot access the texture being rendered into within pixel shaders for reasons of concurrency and hence two pairs of textures were maintained, one pair for the velocities and one pair for the vertex positions. At each step, one member of the pair would reflect the 'current state' and the other would be written into representing the 'new state'. At the end of the simulation step their rôles would be reversed.

The first part of figure 9.12 shows the shader used to sum forces. The force on each vertex is calculated as the sum of forces due to Hookian springs which connect it to its immediate neighbours. The integration of force over time to give velocity is simply Euclid's method due to its ease of implementation and compactness; space is a premium in shaders.

The first part of figure 9.13 shows the shader used to update vertex positions. Again, the velocity is integrated by simple summing to make the

```
1  // Shader to calculate force on each vertex
2
3  #include "map.cg" // Tools to map to and from deformed space
4
5  // Various 'tunables' to change dynamics
6  #define SPRING_CONST 1.5
7  #define GRAVITY -0.015
8  #define FRICTION 0.8
9  #define DAMPING 0.98
10 #define STEP 0.075
11
12 // Texture samplers for reading current vertex positions and velocities.
13 uniform const samplerRECT verticesTex : TEXUNIT0;
14 uniform const samplerRECT velocitiesTex : TEXUNIT1;
15 // Dimensions and ideal inter vertex spacing of cloth
16 uniform const int2 dimensions;
17 uniform const float idealSpacing;
18
19 // Add the cloth vertex/cloth vertex Hooke's law forces to a particular vertex.
20 void update_force(const int2 dPos, const int2 wPos, const float3 v, inout float3 force) {
21   float3 delta;
22   float d;
23   int2 cPos = dPos + wPos;
24   if ((cPos.x < dimensions.x) && (cPos.y < dimensions.y) && (cPos.x >= 0) && (cPos.y >= 0)) {
25     delta = texRECT(verticesTex, cPos).rgb - v;
26     d = length(delta) / length((float2)dPos); d -= idealSpacing;
27     force += clamp(SPRING_CONST * d, -5,5) * normalize(delta);
28   }
29 }
30
31 #define DO_FORCE(a,b) { update_force(int2(a,b), wPos, v, force); }
32 float3 main(int2 wPos : WPOS) : COLOR {
33   float3 v = texRECT(verticesTex, wPos).rgb;
34   float3 vel = texRECT(velocitiesTex, wPos).rgb;
35   float3 force = float3(0,GRAVITY,0);
36   // Resolve spring forces in a cross around the vertex
37   DO_FORCE( 0, -1 ); DO_FORCE( 1, 0 ); DO_FORCE( 0, 1 ); DO_FORCE( -1, 0 );
38   vel = DAMPING*vel + STEP*force;
39   // Now deform so taget object becomes unit-sphere
40   map_pos_to_sphere(v, vel, v);
41   // If we're inside the sphere or on the surface remove the component of
42   // velocity into the surface and attenuate the component tangential.
43   if (length(v) <= 1.0) {
44     v = normalize(v); float d = dot(v, vel);
45     if (d < 0.0)
46       { vel -= d * v; vel *= FRICTION; }
47   }
48   // Deform back to real-space.
49   map_sphere_to_pos(v, vel, v);
50   return vel;
51 }
```

Figure 9.12: The pixel shader for summing the forces on the cloth vertices and removing normal components of velocity.

shader small and fast.

The second part of the simulation takes place after deformation such that the target object becomes the unit sphere. In figure 9.13 the new vertex position is simply moved so that it does not penetrate the object. In figure 9.12 some basic surface physics is performed removing components of velocity into the object and attenuating tangential components to simulate friction.

Both shaders use a Cg implementation of the deformation scheme which is implemented in figure 9.14. Here the components of each key generator are stored in a texture which is used to deform any vertex passed to the function.

Results

Figure 9.15 shows a selection of scenes from the final proof of concept application. You can see the central object which is a unit-sphere which has been deformed by 4 key-generators with linear fall-off. The cloth simulation is then allowed to fall on the object and slides off. All of this is performed on the GPU in real-time (50 frames per second) for a 1024-vertex cloth model.

9.6 Chapter summary

In this chapter we investigated various techniques, based on GA, which might find applications on a Graphics Processing Unit common on many desktop PCs. We briefly discussed the architecture of these units and outlined how they might be 'tricked' into performing general purpose computation. It was mentioned that GA algorithms might well be very 'GPU-friendly'

in that they consist of a number of steps which have few or in many cases no special cases so that the operations may be carried out in parallel using identical processing units.

We then presented a set of samples using these techniques, accelerated via a GPU. A mesh deformation scheme was discussed, based on rotor interpolation, and an example of how rotor interpolation may be performed on the GPU was shown. This solution was benchmarked and shown to be significantly faster than a pure-software approach.

A simplistic physics simulation was also presented running on the GPU. Here the fact that rotor interpolation may be viewed in some way as a 'frame distortion' scheme allowed us to perform surface physics on a deformed sphere rapidly and in a simple manner.

The techniques outlined in this chapter are a useful starting point for future focused research into the hardware acceleration of GA-based algorithms.

```
1  /* Program to calculate new vertices from velocities */
2  #include "map.cg"
3
4  float3 main(in float2 wPos : WPOS,
5                  uniform samplerRECT verticesTex : TEXUNIT0,
6                  uniform samplerRECT velocitiesTex : TEXUNIT1
7                  ) : COLOR
8  {
9    // Get the current vertex position (v) and velocity (vel)
10   float3 v = texRECT(verticesTex, wPos).rgb;
11   float3 vel = texRECT(velocitiesTex, wPos).rgb;
12   float3 junk = float3 (1,0,0);
13
14   // 'Integrate' the velocity to update the vertex position.
15   v += vel;
16
17   // Deform so our target is the unit-sphere.
18   map_pos_to_sphere(v, junk, v);
19
20   // If we are inside the sphere, correct.
21   if (length(v) < 1.0) {
22     v = normalize(v);
23   }
24
25   // Deform back.
26   map_sphere_to_pos(v, junk, v);
27
28   return v;
29 }
```

Figure 9.13: The pixel shader used to update the vertex position and correct for penetration.

```
// ** Utility functions to map to and from a rotor deformed space. **
// The 6D generators representing the deforming rotors are stored in a
// texture. The 6 components are held in a 2-pixel pair where the first pixel
// hold the 3 rotational components and the second holds the 3 positional
// components.

#include "rotor_tools.cg"

#define MAX_GENERATORS 4 // Maximum number of deforming generators stored in texture
#define RANGE 2.5        // The range of influence of the deformation rotors.
const float M_PI = 3.14159265358979323846; // Pi (more or less)
uniform const samplerRECT deformGenerators : TEXUNIT3; // The actual generators are held in texture unit 3.

// Function to deform a point and normal by the deforming rotors
void map_pos_to_sphere(in const float3 pos, inout float3 normal, out float3 sph)
{
  int i;
  sph = pos;
  for(i=0; i<MAX_GENERATORS; i++) {
    float3 rotPart = texRECT(deformGenerators, int2(i*2, 0)).rgb;
    float3 location = texRECT(deformGenerators, int2(i*2 + 1, 0)).rgb;

    if(length(rotPart) > 0.0) {
      sph -= location;
      float ls = length(sph) / RANGE;
      if(ls < 1.0) {
        float3 myrp = -rotPart * (0.5 * cos(M_PI * ls) + 0.5);
        float4 r1;
        if(length(myrp) > 0.0) {
          exp_rot_generator(myrp, r1);
          sph = apply_rot_rotor_to_point(r1,sph); normal = apply_rot_rotor_to_point(r1,normal);
        }
      }
      sph += location;
    }
  }
}

// Function to undo the deformation applied by map_pos_to_sphere().
void map_sphere_to_pos(in const float3 sph, inout float3 normal, out float3 pos)
{
  int i;

  for(i=MAX_GENERATORS-1; i>=0; i--) {
    float3 rotPart = texRECT(deformGenerators, int2(i*2, 0)).rgb;
    float3 location = texRECT(deformGenerators, int2(i*2 + 1, 0)).rgb;

    if(length(rotPart) > 0.0) {
      pos -= location;
      float ls = length(pos) / RANGE;
      if(ls < 1.0) {
        float3 myrp = rotPart * (0.5 * cos(M_PI * ls) + 0.5);
        float4 r1;
        if(length(myrp) > 0.0) {
          exp_rot_generator(myrp, r1);
          pos = apply_rot_rotor_to_point(r1,pos); normal = apply_rot_rotor_to_point(r1,normal);
        }
      }
      pos += location;
    }
  }
}
```

Figure 9.14: The vertex shader utility functions for mapping to and from a rotor-deformed space.

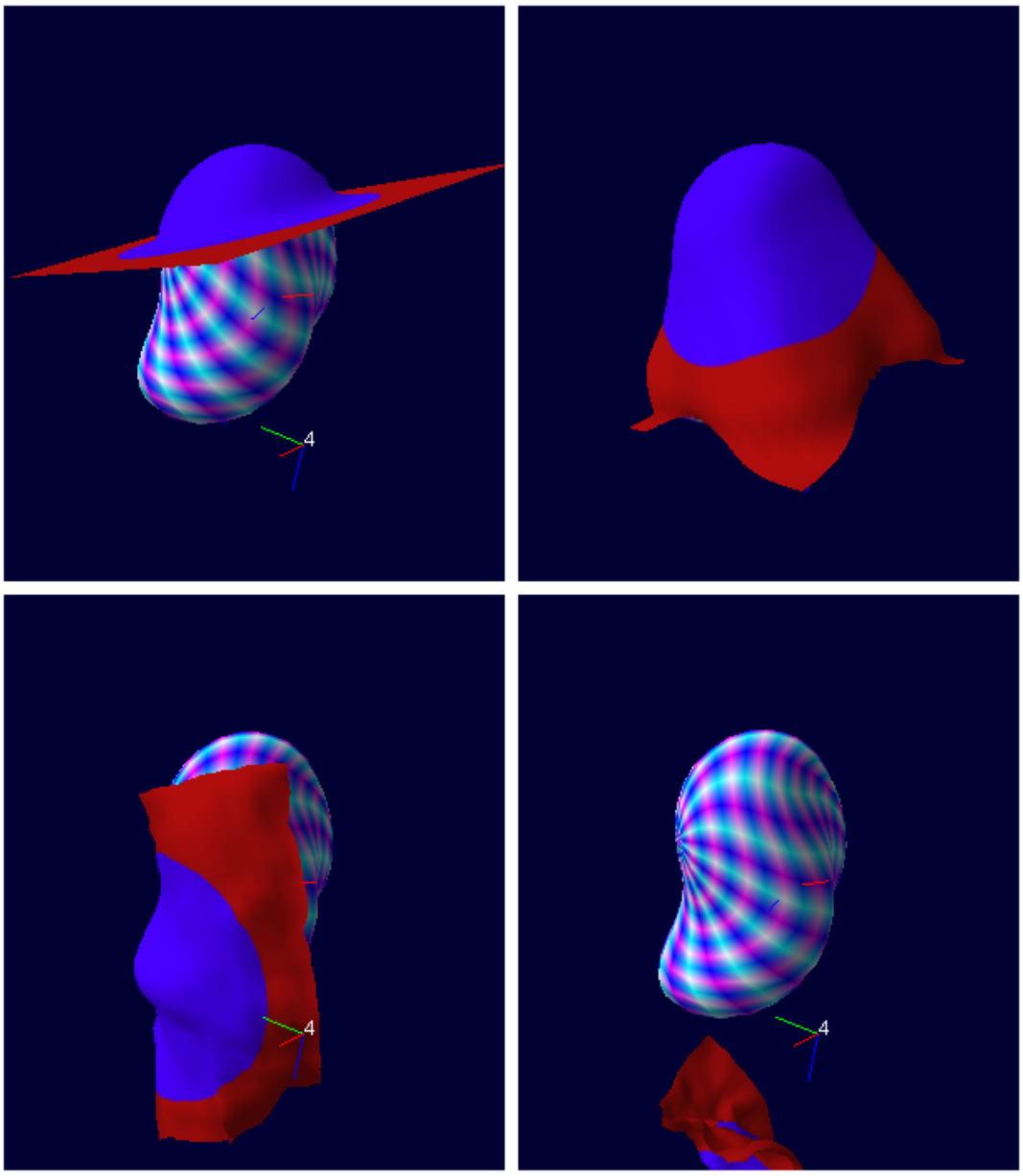


Figure 9.15: A selection of scenes from the penetration demo showing the simulation of a simple cloth model on the surface of a deformed sphere.

10

Conclusions and Future Work

In this chapter we shall collate all of the findings from the previous chapters and give them a context in relation to each other. Future applications for the various findings will also be discussed.

10.1 Review of Achievements

In this section we briefly review the achievements and findings from each chapter.

10.1.1 Non-Euclidean geometries

In chapter 5, a framework for extending the conformal model to deal with non-Euclidean geometries was shown, with particular emphasis on hyperbolic geometry. It was shown that the geometry represented by a model is entirely determined by the choice of null-vector representation and rotors. The pure-rotation and pure-translation rotors for hyperbolic space were de-

rived and it was shown that from them the usual distance metric for hyperbolic geometry could be obtained.

Already some work using the conformal model to represent non-Euclidean geometry has found application in cosmology[33] leading, potentially, to important insights on our universe.

10.1.2 Fractals

In chapter 6 an extension to complex numbers, similar to that of quaternions, was developed for arbitrary dimension. It was noted that, in GA, quaternions are simply special cases of a wider variety of algebras. This extension was used to form a dimension-agnostic formulation for the classic complex iteration-based Julia and Mandelbrot fractal sets. In addition an existing distance estimation formula was shown to be valid using this extension allowing for the ray-tracing of arbitrary dimension sets.

Real-world applications of fractals are notoriously difficult to find but the opening up of escape-time fractals to non-Euclidean geometries provides a number of opportunities for 'recreational mathematics' and the generation of attractive images.

10.1.3 Rotor exponentiation

In chapter 7 it was hypothesised that all the rotors we used in the conformal model could be obtained by exponentiating a corresponding generator bivector the components of which would be geometrically meaningful. A

closed form solution for *both* the exponentiation and subsequent inverse exponentiation (modulo the identification of rotations by $2n\pi$) was derived.

From this an algorithm for directly mapping the components of the generator to a 4×4 matrix suitable for use in existing graphical pipelines was developed. A matching algorithm for directly converting a matrix to a generator, again identifying rotations of $2n\pi$, was also developed.

This particular chapter has almost limitless application. Not only is the linear space of the bivectors mapped to the non-linear space of rigid-body transformations but the appropriate inverse mapping was also defined. Using this method many existing linear optimisation algorithms or interpolation schemes could be extended to deal with rotation and translation *simultaneously*.

10.1.4 GPU-based techniques

In chapter 9 the techniques developed in chapter 7 were implemented on the programmable portion of modern Graphics Processing Units. Such *shaders* were used to develop sample graphics algorithms which made use of the mappings developed in this thesis.

Specifically simple mesh deformation and collision detection examples were show. The examples demonstrated that not only was GA a natural language for developing such algorithms allowing one to use much geometric insight but they were also compact enough to program so that they could be efficiently implemented in hardware.

10.2 Future work

Of all the work presented in this thesis perhaps that with the clearest scope for future work is the work on rotor exponentiation. The ability to map rotors conveniently into a 6d linear space allows a great deal of algorithms initially developed for translation and points to be converted into algorithms acting on rotations. In addition the freedom to move wherever one wishes within this space coupled with a well defined 'distance' between rotors (letting the components of the bivector be those of a 6d vector and using the normal Euclidean distance) allows one to investigate minimisation algorithms for fitting rotations to data.

Already, unpublished work has shown promise for this approach in animation interpolation and compression using motion capture data and further work will hopefully be fruitful in this area.

Bibliography

[1] A. Aouady. Julia sets and the mandelbrot set. In H.-O. Peitgen and D. H. Richter, editors, *The Beauty of Fractals: Images of Complex Dynamical Systems*, page 161, Berlin, 1986. Springer-Verlag.

[2] C. Bajaj, H. Lee, R. Merkert, and V. Pascucci. NURBS based B-rep Models from Macromolecules and their Properties. In C. Hoffmann and W. Bronsvort, editors, *Proceedings Fourth Symposium on Solid Modeling and Applications, Atlanta, Georgia*, pages 217–228. ACM Press, 1997.

[3] M. F. Barnsley and L. P. Hurd. *Fractal Image Compression*. AK Peters, Wellesley, MA, USA, 1993.

[4] M. F. Barnsley, A. Jacquin, F. Malassenet, L. Reuter, and A. D. Sloan. Harnessing chaos for image synthesis. *Computer Graphics*, 22(4):131–140, Aug. 1988.

[5] M. F. Barnsley and H. Rising. *Fractals Everywhere*. Academic Press Professional, Boston, 1993. ISBN 0120790610.

[6] C. Blanc and C. Schlick. Accurate parametrization of conics by nurbs. *Computer Graphics and Applications*, 16(6):64–71, November 1996.

[7] D. A. Brannan, M. F. Esplen, and J. J. Gray. *Geometry*. Cambridge University Press, 1999. Ch. 6.

[8] B. Branner. The Mandelbrot set. In R. L. Devaney and L. Keen, editors, *Chaos and Fractals: The Mathematics Behind the Computer Graphics*, volume 39, pages 75–105, Providence, RI, 1989. Amer. Math. Soc.

[9] S. Buss and J. Fillmore. Spherical averages and applications to spherical splines and interpolation. *ACM Transactions on Graphics*, pages 95—126, 2001.

[10] Cambridge Astrophysics Group. Cambridge GA library for Maple. `http://www.mrao.cam.ac.uk/~clifford/software/`.

[11] J. Cameron. Applications of Geometric Algebra, August 2004. PhD First Year Report, Cambridge University Engineering Department.

[12] Z. Cendes and S. Wong. C1 quadratic interpolation over arbitrary point sets. *IEEE Computer Graphics and Applications*, pages 8–16, Nov 1987.

[13] W. Clifford. Applications of Graßmann's extensive algebra. *Am. J. Math.*, 26(6):613–627, 1878.

[14] Y. Dang, L. H. Kauffman, and D. J. Sandin. *Hypercomplex iterations: distance estimation and higher dimensional fractals*. World Scientific, 2002. ISBN 9810232969.

[15] C. Doran and A. Lasenby. *Geometric Algebra for Physicists*. Cambridge University Press, 2003.

[16] A. Douady. Julia sets and the Mandelbrot set. In H.-O. Peitgen and D. H. Richter, editors, *The Beauty of Fractals: Images of Complex Dynamical Systems*, page 161, Berlin, 1986. Springer-Verlag.

[17] A. Dress and T. Havel. Distance geometry and geometric algebra. *Foundations of Physics*, 23(10):1357–1374, Oct. 1991.

[18] D. Dunham. Transformation of Hyperbolic Escher Patterns. *Visual Mathematics*, 1(1), 1999.

[19] Euclid. *The Thirteen Books of Euclid's Elements translated from the text of Heiberg*. Dover Publications, New York, 1956. Translated with introduction and commentary by Thomas L. Heath.

[20] K. Falconer. *Fractal Geometry: Mathematical Foundations and Applications*. John Wiley & Sons, Ltd., West Sussex, 2003. ISBN 0470848618.

[21] K. J. Falconer. *The Geometry of Fractal Sets*. Cambridge University Press, 1985.

[22] S. Fontijne, L. Dorst, and T. Bouma. GAIGEN: a Geoemtric Algebra GENerator, 2001. Available for download at `http://carol.wins.uva.nl/~fontijne/gaigen/about.html`.

[23] S. S. H. U. Gamage and J. Lasenby. New least squares solutions for estimating the average centre of rotation and the axis of rotation. *Journal of Biomechanics*, (35):87–93, 2002.

[24] V. Govindu. Lie-algebraic averaging for globally consistent motion estimation. In *Proceedings of CVPR*, pages 684–691, 2004.

[25] H. Graßmann. Der Ort der Hamilton'schen Quaternionen in der Ausdehnungslehre. *Math. Ann.*, 12:375, 1877.

[26] W. R. Hamilton. *Elements of Quaternions*. Longmans, Green, 1866.

[27] W. R. Hamilton. *The Mathematical Papers of Sir William Rowan Hamilton*. Cambridge University Press, 1967.

[28] D. Hestenes. Old wine in new bottles: a new algebraic framework for computational geometry. In E. Bayro-Corrochano and G. Sobcyzk, editors, *Geometric Algebra with Applications in Science and Engineering*, pages 1–16, Boston, 2001. Birkhauser.

[29] D. Hestenes and G. Sobczyk. *Clifford Algebra to Geometric Calculus: A unified language for mathematics and physics*. Reidel, 1984.

[30] D. Hestenes and R. Ziegler. Projective geometry with clifford algebra. *Acta Applicandæ Mathematicæ*, 23:25–63, 1991.

[31] H. Jürgens, H.-O. Peitgen, and D. Saupe. *Chaos and Fractals: New Frontiers of Science*. Springer-Verlag, New York, 1992. ISBN 038797903.

[32] J. Kessenich, D. Baldwin, and R. Rost. *The OpenGL Shading Language, version 1.05*. Feb. 2003.

[33] A. Lasenby. Keynote address SIGGRAPH, 2003.

[34] A. Lasenby, J. Lasenby, and R. Wareham. A covariant approach to geometry using geometric algebra. Technical report, Cambridge University Engineering Department, 2004.

[35] A. Lasenby, J. Lasenby and R. Wareham. A covariant approach to geometry and its applications in computer graphics. Technical report, Cambridge University Engineering Dept., 2002.

[36] J. Lasenby and E. Bayro-Corrochano. Analysis and Computation of Projective Invariants from Multiple Views in the Geometric Algebra Framework. In M. Rodrigues, editor, *Invariants for Pattern Recognition and Classification*, volume 42. World Scientific, 2000. Series in Machine Perception and Artificial Intelligence, 233pp, ISBN 981-02-4278-6.

[37] J. Lasenby, W. Fitzgerald, A. Lasenby, and C. Doran. New geometric methods for computer vision: An application to structure and motion estimation. *International Journal of Computer Vision*, 26(3):191–213, 1998.

[38] H. Li, D. Hestenes, and A. Rockwood. Generalized homogeneous coordinates for computational geometry. In G. Sommer, editor, *Geometric Computing with Clifford Algebra*, pages 25—58. Springer, 2001.

[39] M. Lillholm, E. Dam, and M. Koch. Quaternions, interpolation and animation. Technical Report DIKU-TR-98/5, University of Copenhagen, July 1998.

[40] B. B. Mandelbrot. *Les objets fractals: forme, hasard, et dimension*. Flammarion, 1975.

[41] B. B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman and Co., New York, 1982. ISBN 0716711869.

[42] Microsoft Corporation. *DirectX 9.0 graphics*. Dec. 2002. Available at `http://msdn.microsoft.com/directx`.

[43] Microsoft Corporation. High-level shader language. In *DirectX 9.0 graphics*. Dec. 2002. Available at `http://msdn.microsoft.com/directx`.

[44] M. Moakher. Means and averaging in the group of rotations. *SIAM Journal of Applied Matrix Analysis*, pages 1—16, 2002.

[45] NVIDIA. *The Cg Language Toolkit*. Jan. 2004. Version 1.2.

[46] NVIDIA. Shader model 3.0 unleashed, 2004. Presentation at SIGGRAPH 2004.

[47] NVIDIA. OpenGL driver 1.0-7664 release notes, June 2005.

[48] F. Park and B. Ravani. Smooth invariant interpolation of rotations. *ACM Transactions on Graphics*, pages 277–295, 1997.

[49] H.-O. Peitgen and D. Saupe, editors. *The Science of Fractal Images*. Springer-Verlag, New York, 1988. ISBN 0387966080.

[50] C. Perwass. The CLU Project. Available for download at `http://www.perwass.de/cbup/clu.html`.

[51] L. Piegl. On NURBS: A Survey. *IEEE Computer Graphics and Applications*, 11(1):55—71, Jan 1991.

[52] D. F. Rogers and J. A. Adams. *Mathematical Elements for Computer Graphics*. McGraw Hill, 2nd edition, 1990.

[53] D. F. Rogers and R. A. Earnshaw, editors. *State of the Art in Computer Graphics – Visualization and Modeling*. Springer-Verlag, New York, 1991. pp. 225—269.

[54] B. Rosenhahn. Pose estimation revisited, 2003. PhD Thesis, Technical Report 0308.

[55] B. Rosenhahn, C. Perwass, and G. Sommer. Free-form pose estimation by using twist representations. *Algorithmica*, (38):91–113, 2004.

[56] B. Rosenhahn, C. Perwass, and G. Sommer. Pose estimation of free-form contours. *International Journal of Computer Vision (IJCV)*, 62(3):267–289, 2005.

[57] B. Rosenhahn and G. Sommer. Pose estimation in conformal geometric algebra part I. *Journal of Mathematical Imaging and Vision (JMIV)*, 22, 2005.

[58] R. Rost. *OpenGL 2.0 Overview*. 3D Labs, 2002.

[59] P. J. Schneider. NURB curves: A guide for the uninitiated, March 1996. `http://www.mactech.com/articles/develop/issue_25/schneider.html`.

[60] M. Segal and K. Akeley. *The OpenGL Graphics System: A Specification (Version 1.4)*. OpenGL Architecture Review Board, 2002.

[61] K. Shoemake. Animating rotation with quaternion curves. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 245–254, New York, NY, USA, 1985. ACM Press.

[62] R. Wareham, J. Cameron, and J. Lasenby. Applications of conformal geometric algebra in computer vision and graphics. In H. Li, P. J. Olver, and G. Sommer, editors, *IWMM/GIAE*, volume 3519 of *Lecture Notes in Computer Science*, pages 329–349. Springer, 2004.