UNIVERSITY OF CALIFORNIA

Santa Barbara

VERSOR
Spatial Computing with Conformal Geometric Algebra

A thesis submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

MEDIA ARTS AND TECHNOLOGY

by

Pablo Colapinto

March 2011

The Master's thesis of Pablo Colapinto is approved.

____
Marko Peljhan

____
Curtis Roads

____
JoAnn Kuchera-Morin

____
Tobias Hollerer

March 2011

Abstract

This Master's thesis investigates new computer graphics synthesis techniques made possible by the Euclidean, spherical, and hyperbolic transformational capacities of conformal geometric algebra [CGA]. An explication of the mathematical system's geometric elements is followed by documentation of *Versor* – an integrated CGA software library for immersive 3D visualizations and dynamic simulations.

Contents

1 Introduction:

No attention should be paid to the fact that algebra and geometry are different in appearance.

–Omar Khayyám

L'algèbre n'est qu'une géométrie écrite; la géométrie n'est qu'une algèbre figurée.

–Sophie Germain

The present work explicates 5-dimensional conformal geometric algebra [CGA] and documents a new C++ implementation for graphics synthesis. Geometric algebra [GA] is a combinatoric system of spatial logic based on William Clifford's hypercomplex algebras of the 19th century. Holistic, scalable, and filled with "common sense", GA integrates various methods for modelling and engineering dynamic systems. Applications exist in computer vision[4] and graphics[6, 1], neural nets[15], DSP[23], robotics[17], optics [21], particle and relativistic physics[16], and metamaterials research[18].[1] The now classic work by the physicist David Hestenes, *New Foundations for Classical Mechanics* (1986) makes a strong argument for learning this approach by demonstrating the **compactness** of the math, while the most recent text by cyberneticist Eduardo Bayro-Corrochano, *Geometric Computing: For Wavelet Transforms, Robot Vision, Learning, Control and Action* (2010), [2] demonstrates the **expressivity** of its powers for geometric reasoning. Many other references make a similar case: with its *isomorphisms*, geometric algebras encapsulate many other mathematical systems. With its *outermorphisms*, solutions worked out in smaller dimensions can often be extrapolated to higher dimensions. It is an expressive logic that allows intuitive mathematical experimentation across a widening range of disciplines. [3]

Working with the algebra, physicists have developed a *conformal split* mapping of a 3 dimensional Euclidean space G^3 into a 5-dimensional one, $G^{4,1}$, based on Riemannian projection of 3D Euclidean space ($\mathbb{R}^3$) onto a hypersphere. Introduced into the geometric algebra community by Hongbo Li, Alan Rockwood, and David Hestenes in 2001, the conformal model greatly simplifies and generalizes calculations of general rigid body movements in Euclidean space. As a mathematical system for describing closed form solutions within Euclidean, spherical, and hyperbolic geometries, conformal geometric algebra opens the door to a rich set of *Möbius*

[1] For a good overview of current research in engineering and graphics see also [14].

[2] see also 2001's *Geometric Algebra with Applications in Science and Engineering*, eds E. Bayro-Corrochano and G. Sobczyk and 2005's *Handbook of Geometric Computing*, eds E. Bayro-Corrochano

[3] A rigorous morphological examination of biological forms expressed in the language of GA should be pursued by embryologists. The *orientability* of the algebra is useful for describing *chiralities* or handedness, which organic systems are particularly sensitive to. I should note an intriguing set of articles written by C. Muses in the late 1970's which point to hypercomplex numbers as having a unique use in modelling biological systems. Published under the auspices of the mobile and mysterious "Research Centre for Mathematics and Morphology," Muses' articles – such as the 1979 *Computing in the Bio-Sciences with Hypernumbers: A Survey* – are filled with parametric lobes and coils, and argue for the specialized use of imaginary numbers in the study of biological form. GA seems poised to contribute significantly to the modelling of such dynamic systems and structures.

transformations typically restricted to the 2D plane. The operators which *reflect, rotate, translate, twist, dilate,* and *boost* other elements and objects are called *versors*.

1.1 Problem Statement

1.1.1 Synthesis Techniques

The use of geometric algebra for exploration in graphic modelling remains a relatively esoteric exercise in the larger scientific and artistic community. As a result, many of its formal characteristics and exotic morphological powers have yet to be explored. Hidden in the algebra are forms whose organic and mysterious characteristics could bear more formal investigation. Great research has been done demonstrating its powers of synthesis, especially with regards to forms and control systems generated by twist groups and motor algebras [19, 24], but hardly the same can be said for tangent groups and hyperbolic boosts, where the literature remains focussed on 2D constructions or coordinate-based methods, and the surface has barely been scratched (or *bent*...). Artists, scientists, designers, and engineers all stand to benefit from the increased spatial intuition these explorations support, and the door they can offer to other domains. From emerging engineering tasks in biometa- and nano- materials to artistic experiments in immersive environments and interactive installations, the modelling of dynamic systems in these domains could witness an acceleration of innovation through the sharing of new CGA synthesis techniques across discplines.

1.1.2 Multimedia Applications

The specific 5D conformal model discussed here was introduced very recently (2001) – and computer graphics applications that leverage its expressivity are few. Because of the challenges in implementing an efficient system (see section 6.2), and the steep curve in re-learning physics in terms of the algebra, these applications have not yet been fully integrated into high-performance multimedia applications, which would allow researchers to engage with the algebra in diverse ways. Most applications emphasize scripting techniques which help users learn the algebra, or optimize its implementation – efficient CGA software complete with flexible data flow and signal processing capabilities, user interfaces for fast prototyping, and built-in dynamics and camera navigation are rare. A language of extension and orientation of form and space, GA is an environment which must be entered to be understood. Needed are more applications designed for real time immersive environments that allow researchers to realize novel multimodal experiences.

1.1.3 Pedagogy

Even with a growing base of adopters, geometric algebra is still largely undertaught, and only a handful of schools[4] specialize in its advancement. While there are a few

[4]Cognitive Systems at Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik: http://www.ks.informatik.uni-kiel.de; Cambridge University Geometric Alge-

excellent textbooks and dissertations[5] available introducing the logic of GA, the topic of *hypercomplexity* itself remains distant to many multimedia systems engineers. As academia turns towards the quantum and cellular scales of the nanoworld, increased collaboration between biologists, programmers, and physicists will benefit from a common language for spatial dynamics. Modern day graphics systems are an amalgam of matrix, vector, and tensor algebras, a fast but jerry-rigged system which makes it difficult to build higher dimensional intuition and obstructs the typical graphics-based experimenter from making the leap into quantum calculations or nth-order dynamics. Hestenes and others argue that the logical system of geometric algebra naturally facilitates this transition for students.[6]

1.2 Goal of the Present Work

1.2.1 Text

This paper advances the pedagogical program of geometric algebra by de-mystifying its logic. The synthesis of shapes and environments is accompanied by a strategy for understanding these shapes and environments. Some of the explicatory work below, particularly Sections 4 and 5 which anatomize various elements of CGA, follows the textbook *Geometric Algebra for Computer Science* by Leo Dorst, Stephen Mann, and Daniel Fontijne. Sections 2 and 3 draw mostly from the fundamentally intuitive paper *Geometric Algebra Primer* by Jaap Supter (available online), and clear essay *Generalized Homogeneous Coordinates for Computational Geometry* by Hongbo Li, David Hestenes, and Alan Rockwood, as well as various texts by Eduardo Bayro-Corrochano on the conformal model. All of these works have good introductions to the spaces discussed in this thesis. The particular introduction developed here offers some new techniques for form building, such as affine combinations of point pairs and fluid-like tangent fields. Elaborating on methods presented in the textbook by Dorst et al [6], an outline of the implementation of *Versor* follows the formal investigation. Strategies from other texts are also included and referenced in an ef-

bra Research Group: http://www.mrao.cam.ac.uk/~clifford/; Centre for Image Technology and Robotics: http://www.citr.auckland.ac.nz; CINVESTAV in Guadalajara; Department of Physics in Arizona State University: http://geocalc.clas.asu.edu/; Embedded Systems and Applications Group, University of Technology, Darmstadt, Germany; Intelligent Systems Lab University of Amsterdam: http://www.science.uva.nl/research/isla/; Geometric Modeling and Scientific Visualization Research Center, King Abdullah University of Science and Technology, Saudi Arabia.

[5] see the dissertations by D. Fontijne, D. Hildenbrand, R. Wareham

[6] The pedagogical bent of those who promote geometric algebra cannot be overemphasized. Building a common language was the main motivation for William Clifford's research, as it was David Hestenes', who took up the mantle of Clifford's discoveries and served as their torch-bearer through the late 20th century. He continues in Clifford's tradition of battling errors of 'common sense'. In a 1985 essay called "Common Sense Concepts About Motion", co-written with Ibrahim Abou Hallouna, Hestenes classifies the various misunderstandings of college students regarding force, creating a taxonomy of bad intuition. Geometric algebra, one presumes, could be employed to help all that (though it is not mentioned in that text). This notion of *thinking the right way* is inextricably tied to the primary aims of geometric methods, and was a cornerstone of Descartes philosophy when he developed his coordinate system in an appendix to *Discourse on Method*. Since its resurgence, Geometric algebras have been constantly referred to as a "better" way to think about spatial relationships – e.g. a 2001 lecture by nuclear engineer Timothy Havel was called "Geometric Algebra: Parallel Processing for the Mind".

fort to build a clear and concise picture of the mathematical system.

1.2.2 Code

Versor is a C++ graphics-synthesis toolset for exploring new techniques in the manipulation of virtual forms and the activation of dynamic environments. It implements conformal geometric algebra through an efficient and integrated multimedia platform, low-level enough for “serious” applications and high-level enough for “user friendly” functionality. It simplifies experimental exploration in a range of contexts, puts some of the more exquisite features of CGA into the hands of digital practitioners, and can be used for both artistic and engineering-based design. Integrated with various dynamic solvers (including Verlet integration and semi-Lagrangian Navier-Stokes methods), a graphics user interface library (GLV[7]) and an audio synthesis library (Gamma[8]), *Versor* can be compiled as a stand alone application or as an external library. It incorporates many compositional techniques for analysis and synthesis of dynamic forms and structures, some drawn from the CGA research landscape and some introduced for the first time, such as templated “Hyper Fluids” (see Section 6.5.4). *Versor* aims to provide a basis for research requiring the visualization of complex fields such as those found in quantum electrodynamics, gauge theories, lorentz fields, curvature tensors, and morphogenetics. Potential uses include live performances, immersive environments, multimodal interfaces, molecular modelling and crystallography, morphological studies, hyperbolic geometry, and dynamic physics simulations.

1.2.3 Form

Using *Versor* to generate figures in the text below, we will explore the shape of the algebra, the shape of the space it represents, and the shape of some of the environments and inhabitants it can generate. The examples explore of *formulations through transformation* and contribute to the *experimental* arena of GA-based synthesis. Focussing on the particular generative power of *twist motors* and *tangent boosts*, it is hoped that such formal explorations could inspire a range of approaches to artistic and scientific modelling, where geometric algebra is still largely unused but likely to be soon, such as multimedia engineering, materials engineering and bioengineering[9].

[7] Graphic Library of Views, an OpenGL user interface library, was developed by Lance Putnam, Graham Wakefield, and Eric Newman at UCSB. Url: http://mat.ucsb.edu/glv/

[8] A generic synthesis library following a data flow model. Developed by Lance Putnam. Url: http://mat.ucsb.edu/gamma/

[9] For a possible hint at the future of geometric algebra in Materials sciences, see Marco Ribeiro’s recent work on “Moving Media” [18] and cloaking.

1.3 Background

1.3.1 History

Towards the end of the 19th Century, William Clifford became interested in creating an algebraic logic that could fully express the basic spatial concepts: magnitude, direction, area, volume. An expert at extrapolating abstract principles from simple concepts,[10] Clifford built a logic of restoration and reduction (an *algebra*) that fused Hermann Grassman's associative algebra of extensions (which builds higher dimensional vector spaces from lower ones) with William Hamilton's invertible algebra of rotations (the quaternions, which can spin vectors in 3D space). Because of their ability to encapsulate both dimension and transformation, Clifford referred to his algebras as *geometric algebras*.[11]

Clifford died young, and his plan for a unified language for spatial relationships was not fully realized: at the end of the 19th century, the battle for a mathematical language of physics was fought between followers of Hamilton's quaternions and of Josiah Willard Gibbs' vector analysis. Gibbs (and Heaviside) won out, and high schools and universities today teach physics using the language of vectors.[12] Still, because of their usefulness in the spin-centric world of quantum physics, Clifford geometric algebras were eventually developed into a geometric calculus flexible enough to be used within a variety of scientific scales and disciplines, from quanta of photons to gravitational fields. In the search for symmetries of the physical world – properties that are invariant under certain transformations – geometric algebras have proven irresistably coherent systems for modelling the natural laws.

[10]For an eye-opening example of Clifford's particular form of thinking see *The Common Sense of the Exact Sciences*, published after his death in 1879. Clifford constructs his explications in a manner that belies how they become complicit in his algebra – that is, first by a chapter on Number, then on Space, Quantity, Position, Motion and Mass. The work remains unfinished, partially completed by editors, and the chapter on Mass does not exist. Still, that Clifford's systematic approach to thinking is mirrored in the structure of his algebra is obvious and elegant. This measured and designed logic behind his mathematical structure was the result of Baconian methods - an attempt to articulate exactly the *phenomenon* of spatial relationships.

[11]Today's geometric Aagebra is more specifically a *nondegenerate* subset of the much more general Clifford Algebra, though some authors prefer an more general understanding of GA as a method of representation: for instance, Rida Farouki (at UC Davis) writes, "Informally, we may consider any space whose elements are subject to sum and product operations as constituting a geometric algebra, if the operations admit simple geometrical interpretations. Thus the first geometric algebra was probably the practice, in ancient Greece, of regarding products of two and three numbers as areas and volumes."[11]Of course, formalized algebra was organized much later by al-Kwharizmi, and the logical union of geometrical concept with algebraic symbol has taken millenia for humans to master. We are still confounded by it.

[12]The term "vector" itself comes from Hamilton's notion of a "pure" quaternion. See footnote 19.

1.3.2 Characteristics

Geometric Concept	**Algebraic Representation**	**Dimension**
Magnitude	Scalar	0
Direction	Vector	1
Area	Bivector	2
Volume	Trivector	3

Table 1: Basic Elements of Geometric Algebra in $\mathbb{R}^3$

Geometric algebras are built with a few simple rules but possess powerful combinatoric properties which facilitate both analysis and synthesis of n-dimensional spaces. They combine vector, matrix, and tensor algebras in one algebraic system. [13] Physicists point to its "unifying" capacities in this regard. [14]

Complex systems – algebras included – are often built from collections of simple rules. Modern physical sciences rely upon the assumption that much of the universe's demonstrated complexity can be coordinated by just a few symbols. Symbolically, geometric algebra is not very complex at all, and its axiomatic logic is clear and concise, like a good algebra should be. As such, it operates at a high level of geometric abstraction when developing equations.

GA provides a mathematical structure for operating with a logic of *transformations*, or *perturbations*. Below is summary of some of its nice expressive benefits:

Isomorphisms: Tensor, vector, and matrix algebras are all embedded through characteristics shared with group theory and lie algebras. Complex numbers, Plücker coordinates, Dirac and Pauli Matrices, and the symmetries of various particles as described by lie groups and their algebras have been shown to be isomophic to, and easily represented by, various metrics of GA. These isomorphisms are critical to the successful promotion of GA; different fields of science could potentially use the same mathematic symbols, bridging gaps between quantum physicists and biologists, for instance.

Outermorphisms: Discoveries made with simpler elements in lower dimensions can often be generalized to higher dimensions. The logic is designed to be extrapolated meaningfully, and the outermorphic properties of the algebra allow for this. This helps in building intuition and experimenting with algorithms.

[13]One might loosely argue that this integration advances aims to systematize the triplet symmetries of the physical world: *Force (Vector)$\mapsto$Structure(Matrix)$\mapsto$Function(Tensor).* Some other mathematical structures that are closely related include Lie algebra–which manipulates groups of small perturbations to create continuous transformations – and combinatorics, which analyzes numerical permutations. Purely mathematical analyses of geometric algebras also usually involve concepts from group theory, and it has been shown to embed more specific concepts like screw theory, gauge theory, and Boolean logic. These parallels come from far-reaching *isomorphic* properties of the system.

[14]For instance, it is currently being adapted for head-to-foot robot control, artificial intelligence and computer vision by Eduardo Bayro-Corrochano of Cinvestav in Guadalajara. One should note there is also a U.S. patent (by Hestenes and Rockwood) out on the use of conformal geometric algebra for robotic applications . . . though an examination of what exactly that limits would be an interesting paper in itself. The conformal mapping is, at its core, just a quadratic equation.

Automorphism: Some elements within the algebra, known as versors or rotors or spinors, can be used to compose transformations upon other elements. These versors form a closed automorphic group, such that multiplying them together will return another member of the group. This powerful structure allows for transformations to be concatenated through multiplication as is done in matrix algebra. As we will see in exploring the conformal model, the complete set of Euclidean and conformal transformations are possible with the versor construction: involutions, inversions, translations, rotations, screw motions, dilations, and transversions.

Diffeomorphisms: The *general covariance* of GA allows solutions to problems to be solved outside any particular coordinate system: natural laws can be thought of in their purely geometric form and then placed within some arbitrary coordinate system afterwards.

Dualities: Calculating relationships between dual representations is a consistent, intuitive, and simple process. To calculate the (Hodge) dual, one divides out the enclosing space (multiplies by the inverse of the pseudoscalar tangent space). Similarly, because of duality, multivectors can be defined directly or indirectly. Many simplifications to complex problems are provided by duality, specifically in complex analysis of vector fields (and, now, higher dimensional fields).

Perturbations: The aforementioned automorphic group of versors can be generated from two-dimensional *bivectors*, which can be linearly added and weighted, allowing complex twisting motions to be interpolated using common Bezier techniques. This introduces a novel differential calculus of continuous deformations that is isomorphic to the Lie algebras, yet more easily navigated.

Representations: Elements of the algebra serve as both operators and objects in a consistent and predictable way. One can, for instance, construct a perturbation operator that is itself perturbed by another operator. This greatly simplifies dynamic modelling by providing a straight forward linear approach to creating higher order phenomena.

Chirality: The anticommutivity of the algebra powers its *orientability*. Elements beyond simple vectors have directions: circles spin clockwise or counterclockwise; screws are right-handed or left-handed. This language of symmetrical properties makes GA particularly suitable for describing polarity-sensitive phenomena, such as biomolecular interactions and bianisotropic materials.

5D conformal geometric algebra has its own particular characteristics

- Euclidean, homogenous, and conformal spaces are contained in one algebraic system. A simple substitution of the basis representation of infinity allows for simultaneous hyperbolic (negatively curved), spherical (positively curved), and Euclidean (straight) spaces. This enables multiple ways to design, navigate and "carve up" a space. Many applications benefit from working in alternatively curved spaces.[15] For instance, in Euclidean geometry, the number of polygons that regularly tessellate a space is limited. This is not the case in hyperbolic (negatively curved) space.

- Rich expression through disambiguation of the conceptual difference between a point in space and vector in space, and different representations for different kinds of vectors. The 3D Cartesian system "works" by using vectors to describe points in space, confusing the two, since it considers vectors as both forces and locations. In the 5D conformal model of space, the relationship is precisely clarified: points are *null vectors*. Non-null vectors themselves can represent *directions*, *tangents*, or *normals*, and each type of vector behaves differently under various transformations.

- The inner product between two points returns the square of their Euclidean distance. This is very natural for describing the basic properites of many physical fields, for instance, since the inner product of any point with another point returns a vector potential. It also makes sense in a more general signal processing sense – the square of deviations are what are summed.

- Useful geometric entities such as spheres and circles are introduced as basic elements of the algebra, in addition to vectors. Furthermore, these entities can be real or imaginary.

- Novel relationships can be explored, such as the sphere orthogonal to two other spheres[16]. Some of these novel features present new ways of analyzing statistical properties of a system, and have implications for topological modelling.

[15] The Universe is one such application . . .

[16] Dorst et al have termed such orthogonal round elements *plunges*. They are a dual representation of the *meet*.

2 Geometric Products

We now proceed to do something which must apparently introduce the greatest confusion, but which, on the other hand, increases enormously our powers.

-William Clifford

The GA system of spatial relationships is orchestrated by three basic operations: the **inner**, **outer**, and **geometric** products. Here we present these combinatoric aspects of geometric algebra relevant to our discussion.[17] A *space* is *spanned* by basis vectors - in 3D typically expressed as x, y, z axes. Here we write with an e_1, e_2, e_3 basis notation. Some authors write σ_1, σ_2, σ_3.

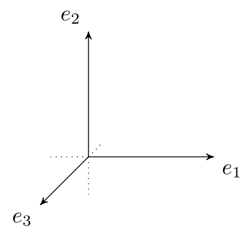


Figure 2.1: Basis 1-blades e_1,e_2, and e_3 in G^3 represent directed magnitudes (e.g. x, y and z). Linear combinations of these basis blades define a vector: $\mathbf{v} = \alpha e_1 + \beta e_2 + \gamma e_3$.

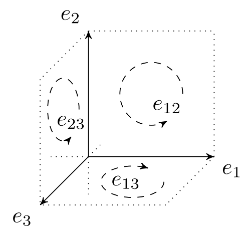


Figure 2.2: Basis 2-blades e_{12},e_{13}, and e_{23} in G^3 represent directed unit areas. Linear combinations of these basis blades define a bivector: $\mathbf{B} = \alpha e_{12} + \beta e_{13} + \gamma e_{23}$.

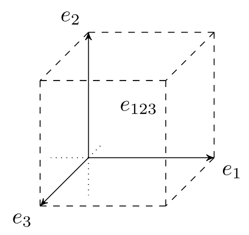


Figure 2.3: The basis trivector e_{123} in G^3 is also known as the pseudoscalar I. As the highest grade blade I is sometimes referred to as the *tangent space*. Multiplying elements by I^{-1} (i.e. dividing out the tangent space) returns their *dual* representation. In G^3 the bivectors and vectors are dual to each other: $vI^{-1} = B$ and $BI^{-1} = v$. Typically one writes $v^* = B$ and $B^* = v$.

Figures 2.1 through 2.3 depict the various elements of an orthononormal *frame* that spans a 3-dimensional space. The frame is comprised of three basis *vectors* e_1, e_2, e_3 depicted in Figure 2.1. Figure 2.2 depicts the various *bivectors* contained within the space. Figure 2.3 depicts the *trivector* that is defined by the space itself.

[17]For several good introductions to to the logic of GA please see the references [7, 6, 10]. Additionally, papers on Clifford algebras, quaternions, spinors, lie algebras, inversive geometry, complex analysis all apply here and can help round out the discussion.

Input a	**Input b**	**Output**
1	0	1
1	1	0
0	0	0

Table 2: The bitwise `XOR` “truth table” measures similarity of inputs.

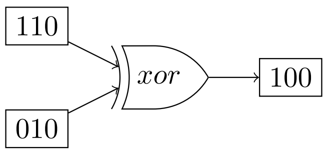
blade	**bits**	**grade**
α	000	0
e_1	001	1
e_2	010	1
e_3	100	1
e_{12}	011	2
e_{13}	101	2
e_{23}	110	2
e_{123}	111	3

Table 3: Binary Representation of Basis Blades in G^3

These elements can be combined through three basic operations: the inner, outer and geometric products.

To understand these operations, it helps to think digitally. Daniel Fontijne made a fundamental connection between a particular simple digital binary operation and the *geometric product.* Let’s take a look at this binary operation `XOR`: the *exclusive-or.*

It is commonly diagrammed as this:



Two binary numbers, `110` and `010` are compared *bitwise.* That is, first we compare the right most bits, then the middle bits, then the left most bits. For each pair, we are only interested in whether the bits are different or the same. If they are different, we return a `1`. If they are the same we return a `0`. Bitwise operations can be presented in a truth table:

Now, consider Table 3. The first thing to notice is that with three bits we can fully represent the different *graded* elements of three dimensional space, as illustrated in in Figures 1 - 3. That is, the different combinations of 1s and 0s of three bits *represent* the various basis elements of our three dimensional space. Every possible combination of "on" and "off" coincides with a particular basis blade. The number of 1s in each digital representation tells us the *grade* of the blade.

In the language of mathematics, we say an n-dimensional vector space V^n generates a "graded" algebra G^n (sometimes written G_n or Cl^n) which is composed of dimensions 0 through n.

A set of n linearly independent *1-blades* (e.g. x, y, z or e_1, e_2, e_3 or σ_1,σ_2,σ_3) form a *basis* that span a *space* (e.g Euclidean $\mathbb{R}^3$). These basis elements can be combined to form *2-blades*, etc, up to *n-blades*. *Subspaces* of dimension $n-1$ are linear combinations of basis blades weighted with coefficients $\alpha, \beta, \gamma, \ldots$. We are currently most familiar with the 1D subspace of *vectors*, though as we'll see higher *graded* elements exist naturally within the algebra. Thus, in $\mathbb{R}^3$, with basis blades e_1, e_2, e_3, a vector $\boldsymbol{a}$ can be written:

$$\boldsymbol{a} = \alpha e_1 + \beta e_2 + \gamma e_3 \tag{2.1}$$

More generally, a vector in $\mathbb{R}^n$:

$$\boldsymbol{a} = \vec{v} = a^i e_i = \sum_{i=1}^{n} s_i e_i \tag{2.2}$$

Such a vector is a *subspace* of *grade* 1, or a *1-blade*. Grade 0 elements of no dimensionality also exist within the space – these are the scalar numbers α, β, γ,etc. The highest dimensional space represented in the set isn't really a subspace but rather a *tangent space* and its representation is called the *pseudoscalar*. It is expressed with the symbol I. In $\mathbb{R}^3$, with basis e_1, e_2, e_3, the pseudoscalar I is a 3-blade, and so is of grade 3 and can be identified thus:

$$I = \bigwedge_{i=1}^{n} e_i = e_1 \wedge e_2 \wedge e_3 = e_{123} \tag{2.3}$$

The wedge product used here comes from Grassman's typically overlooked calculus of extensions[18], wherein elements of a higher dimensionality can be spanned by elements of a lower dimensionality. It is possible to write

$$e_i \wedge e_j = \begin{cases} e_{ij} & i \neq j \\ 0 & i = j \end{cases} \tag{2.4}$$

in which case two different grade 1 elements wedge together to form a new grade 2 element called a *bivector*, which in $\mathbb{R}^3$ is a 2-blade (in higher dimensional spaces, not all grade 2 elements are 2-blades). Notice that if the two wedged elements are the same, the result is 0, as a blade cannot be wedged with itself. Most critical to the logic of the whole geometric algebraic system is that the wedge (sometimes called the *outer* or *exterior*) product is anticommutive, such that

$$a \wedge b = -b \wedge a \tag{2.5}$$

[18] Dorst et al refer to Forder's 1941 *The Calculus of Extension* for a thorough examination of this extrapolation technique.

which introduces the powerful notion of orientability to our mathematical understanding of space, and which distinguishes geometric algebra from algebraic geometry. Vectors are directed magnitudes, they point left or right. Similarly, bivectors are directed areas - they are made clockwise or counterclockwise depending on the order ($a \wedge b$ or $b \wedge a$) used to created them. Trivectors, like the pseudoscalar I of $\mathbb{R}^3$, also inherently spin in one of two directions.

It follows for $\mathbb{R}^3$ that

$$e_{12} = e_1 \wedge e_2 = -e_2 \wedge e_1 = -e_{21} \tag{2.6}$$

$$e_{13} = e_1 \wedge e_3 = -e_3 \wedge e_1 = -e_{31} \tag{2.7}$$

$$e_{23} = e_2 \wedge e_3 = -e_3 \wedge e_2 = -e_{32} \tag{2.8}$$

and so swapping the order of neighboring numbers in a base is possible if accompanied with a multiplication by -1. We can see this used, for instance, in showing how the geometric product gets reduced after distribution: Multiplication of vectors $\boldsymbol{a} = \alpha_1 e_1 + \beta_1 e_2 + \gamma_1 e_3$ and $\boldsymbol{b} = \alpha_2 e_1 + \beta_2 e_2 + \gamma_2 e_3$ can be distributed the usual way as:

$$\boldsymbol{a} * \boldsymbol{b} = \overbrace{\alpha_1\alpha_2 + \beta_1\beta_2 + \gamma_1\gamma_2}^{symmetric\ part} + \underbrace{(\alpha_1\beta_2 - \beta_1\alpha_2)e_{12} + (\alpha_1\gamma_2 - \gamma_1\alpha_2)e_{13} + (\beta_1\gamma_2 - \gamma_1\beta_2)e_{23}}_{asymmetric\ part} \tag{2.9}$$

$$\boxed{a * b = \underbrace{a \cdot b}_{symmetric} + \underbrace{a \wedge b}_{asymmetric}} \tag{2.10}$$

which is a *quaternionic spinor*, a group of 4 terms composed of a symmetric (*inner*, *interior*, or *commutative*) scalar product of grade 0 and asymmetric (*outer*, *exterior*, *anti-commutative*, or *wedge*) bivector product of grade 2. The reader may recognize the familiar "dot" and "cross" products from vector analysis, the former a measure of the similarity or parallelism between two vectors and the latter a measure of the distinguishability or othogonality of the two vectors. Both concepts are contained in the geometric product. [19]Such linear combinations of subspaces of various grades are known as *Multivectors*. This particular multivector - a scalar plus a bivector - is called a *spinor* or *rotor*. More specifically, the vector product in G^3 of two unit vectors produces a quaternion, a type of rotor or spinor that can be used to generate a 3D rotation. As we will see shortly, in higher dimensional algebras other types of spinors besides the quarternion exist which can generate different transformations.

[19]The cross product from vector analysis is *dual* to the wedge product of two vectors in G^3. In the battle over a common language for Physics, Gibbs and Heaviside's vector analysis decomposed these symmetric and asymmetric parts into separate operations; the term "vector" itself comes from Hamilton's work with quaternions, and comes from the notion of a “pure” quaternion – that is, a quaternion without a scalar part.

$*$	$\boldsymbol{s}$	$\boldsymbol{e_1}$	$\boldsymbol{e_2}$	$\boldsymbol{e_3}$	$\boldsymbol{e_{12}}$	$\boldsymbol{e_{13}}$	$\boldsymbol{e_{23}}$	$\boldsymbol{e_{123}}$
$\boldsymbol{s}$	s	e_1	e_2	e_3	e_{12}	e_{13}	e_{23}	e_{123}
$\boldsymbol{e_1}$	e_1	s	e_{12}	e_{13}	e_2	e_3	e_{123}	e_{23}
$\boldsymbol{e_2}$	e_2	$-e_{12}$	s	e_{23}	$-e_1$	$-e_{123}$	e_3	$-e_{13}$
$\boldsymbol{e_3}$	e_3	$-e_{13}$	$-e_{23}$	s	e_{123}	$-e_1$	$-e_2$	e_{12}
$\boldsymbol{e_{12}}$	e_{12}	$-e_2$	e_1	e_{123}	$-s$	$-e_{23}$	e_{13}	$-e_3$
$\boldsymbol{e_{13}}$	e_{13}	$-e_3$	$-e_{123}$	e_1	e_{23}	$-s$	$-e_{12}$	e_2
$\boldsymbol{e_{23}}$	e_{23}	e_{123}	$-e_3$	e_2	$-e_{13}$	e_{12}	$-s$	$-e_1$
$\boldsymbol{e_{123}}$	e_{123}	e_{23}	$-e_{13}$	e_{12}	$-e_3$	e_2	$-e_1$	$-s$

Table 6: Cayley Product Table for G^3

A subspace of grade k in $\mathbb{R}^n$:

$$\boldsymbol{A} = \sum_{i=1}^{\binom{n}{k}} a_i e_i = a^i e_i \tag{2.11}$$

where $\binom{n}{k} = \frac{n!}{(n-k)!k!}$ which specifies the number of k-blades in an n-dimensional space.

$\cdot$	e_1	e_2	e_3
e_1	1	0	0
e_2	0	1	0
e_3	0	0	1

Table 4: $\mathbb{R}^3$ Inner Products

$\wedge$	e_1	e_2	e_3
e_1	0	e_{12}	e_{13}
e_2	$-e_{12}$	0	e_{23}
e_3	$-e_{13}$	$-e_{23}$	0

Table 5: $\mathbb{R}^3$ Outer Products

$$e_1 \cdot e_1 = e_2 \cdot e_2 = e_3 \cdot e_3 = 1 \tag{2.12}$$

$$e_1 \cdot e_2 = e_1 \cdot e_3 = e_2 \cdot e_3 = 0 \tag{2.13}$$

Shorthand from Tensor analysis, one borrows the Kronecker Delta Function:

$$e_j \cdot e_k = \begin{cases} 0, & j \neq k; \\ 1, & j = k; \end{cases} \tag{2.14}$$

With this information, we can produce a Cayley Product Table For $\mathbb{R}^3$ (Table 6).

And out of this we can demonstrate that the geometric product of two vectors is a quaternion. More specifically, quaternionic spinors are *ratios* of vectors in $\mathbb{R}^3$.

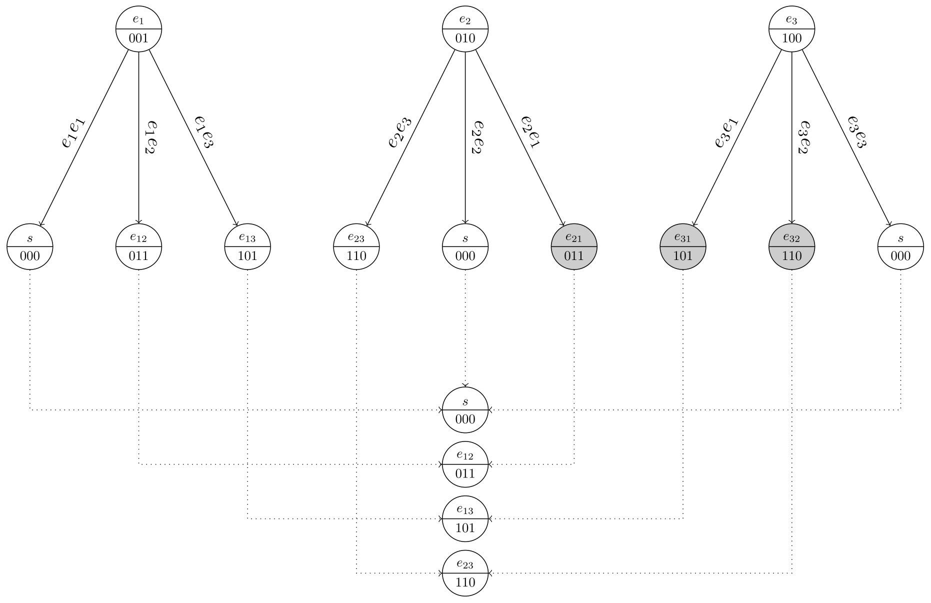


Figure 2.4: Algorithmic anatomy of a quaternion $\mathscr{R} = \frac{\boldsymbol{a}}{\boldsymbol{b}} = \boldsymbol{ab}^{-1} = s + \alpha e_{12} + \beta e_{13} + \gamma e_{23}$. Solid lines represent multiplication. Gray nodes represent instances of a sign change (multiplication by -1). Dotted lines indicate simple summation and reduction of terms.

Product	Symbol	Signification	Relation
inner	$\rfloor$ or $\cdot$	contraction ("dot")	$a \cdot b = \frac{1}{2}(ab - ba)$
outer	$\wedge$	expansion ("wedge")	$a \wedge b = \frac{1}{2}(ab + ba)$
geometric	$*$	ratio	$ab = a \cdot b + a \wedge b$
commutator	$\times$	linear differential	$a \times B = \frac{1}{2}(aB - Ba)$

Table 7: Some Basic Operations in Geometric Algebra

Notice there is no *cross product* per se as there is in vector analysis (the symbol for the cross product instead signifies a commutator product), and the *meaning* of the cross product is supplanted by the extension "wedge" product.[20]

[20]The *cross* product and the *wedge* product are *dual* to each other in 3 dimensions.

3 Minkowskian Metrics

"The paradox is now fully established that the utmost abstractions are the true weapons with which to control our thought of concrete fact."

-Alfred North Whitehead

3.1 The Null Cone

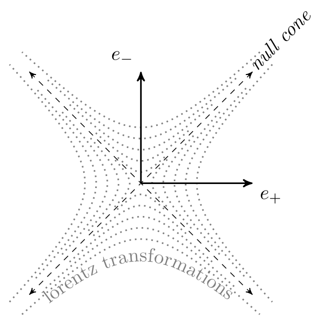


Figure 3.1: The $R^{1,1}$ metric is Minkowskian, with $e_+^2 = 1$ and $e_-^2 = -1$. Rotations in the Minkowski plane $E = e_+ \wedge e_-$ are lorentzian (dotted lines). The "null cone" (dashed lines) is the set of vectors whose directions are invariant under the transformation.

The geometry of the following defining relations are depicted in Figure 3.2. Consider a space where one of the basis vectors squares to -1 instead of 1. This is known as a Minkowski space or Minkowski metric of signature $\mathbb{R}^{1,1}$.

$\cdot$	e_+	e_-
e_+	1	0
e_-	0	-1

Table 8: Minkowski Metric of $\mathbb{R}^{1,1}$

One then creates a *null* basis by defining two new basis elements o and ∞ in terms of e_+ and e_-.

$$\infty = e_- - e_+ \qquad o = .5(e_- + e_+) \tag{3.1}$$

$$e_+ = o - .5\infty \qquad e_- = o + .5\infty \tag{3.2}$$

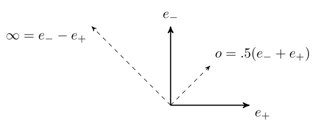


Figure 3.2: A null basis for 2D Minkowski space can be defined through linear combinations of e_- and e_+.

such that $o^2$and ∞^2 both equal 0. This allows the creation of an alternative Minkowski metric:

$\cdot$	o	∞
o	0	-1
∞	-1	0

Figure 3.3: A null metric (degenerate) basis.

As we will explore, this new basis will represent our origin and infinity. In the literature on conformal geometric algebra, o and ∞ are sometimes written as e and $\bar{e}$ or n and $\bar{n}$, and the defining equations with the original basis are not always the same. The end result, however, is usually the same: two basis blades that lie on the null cone.

Transformations in a vector space with Minkowski signature $(n, 1)$ do not behave as they do in a Euclidean metric (see Figure reffig:mnkplane). For instance, in 2 dimensional $\mathbb{R}^{1,1}$ with basis 1-blades e_+ and e_-, rotations of a vector $\alpha e_+ + \beta e_-$on the Minkowski plane E (where $E = e_+ \wedge e_-$) are *Lorentzian.* These basic rotations on the Minkowski plane or E-plane spanned by its basis vectors e_+ and e_- are not elliptical but hyperbolic. [21]

We can combine this Minkowski metric with our Euclidean metric as in Table 9 below,

$\cdot$	e_1	e_2	e_3	e_+	e_-
e_1	1	0	0	0	0
e_2	0	1	0	0	0
e_3	0	0	1	0	0
e_+	0	0	0	1	0
e_-	0	0	0	0	-1

Table 9: A nondegenerate basis.

$\cdot$	o	e_1	e_2	e_3	∞
o	0	0	0	0	-1
e_1	0	1	0	0	0
e_2	0	0	1	0	0
e_3	0	0	0	1	0
∞	-1	0	0	0	0

Table 10: A null basis.

which defines our $\mathbb{R}^{4,1}$ metric by the Minkowski Sum:

$$\mathbb{R}^{4,1} = \mathbb{R}^3 \oplus \mathbb{R}^{1,1} \tag{3.3}$$

Li, Hestenes, and Rockwood refer to this sum as the *conformal split.*[10]

3.2 The Conformal Mapping

With this new basis, one defines a *parameterized* point in space: we take a common Euclidean vector, and add in the two new bases o and ∞. We add a normalized amount of o to homogenize the space $(o = 1)$, and a weighted amount of ∞. This

[21]Known as *Lorentz transformations,* the hyperbolic trajectories they carve out were labelled *world-lines* by Minkowski since they are the virtual representation of the fundamental property of the metric of the space. This is a good example of geometry as described by Klein's Erlangen Program, wherein the fundamental property of a space is defined by *how things move through it,* i.e. with what invariant properties.

critical value of ∞ is set to *one-half the square of the original vector.* Thus our new unique point p is:

$$p = o + \boldsymbol{x} + \frac{1}{2}\boldsymbol{x}^2\infty \tag{3.4}$$

with the Euclidean part written in bold as is the custom in GA texts. Choosing such a quadratic parameterization of the value of ∞ basis blade creates a *conformal mapping* that *projects* or *lofts* our Euclidean space along the new null cone defined and added in Equations 3.1and 3.3 respectively.[22]Equation 3.4 creates a *null* vector in our algebra of $\mathbb{R}^{4,1}$ such that $p \cdot p = 0$. We leave the proof of this nullification to the references[10].

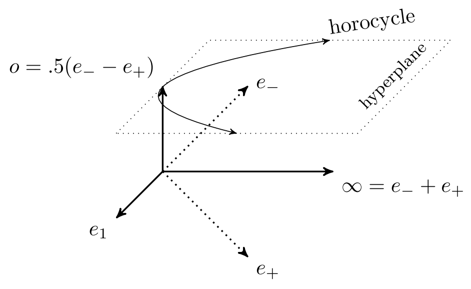


Figure 3.4: The Hyperplane and the Horocycle of $\mathbb{R}^{2,1}$. A vector p in the $\mathbb{R}^{n+1,1}$ metric can be divided into its $\mathbb{R}^n$ and $\mathbb{R}^{1,1}$ components: $p = \mathbf{a_n} + \beta\infty + \alpha o$. Setting α to 1 defines the *hyperplane* of all normalized vectors. Setting β to $\frac{1}{2}\mathbf{a_n}^2$ such that $p^2 = 0$ defines the *horocycle* (if $n = 1$) or *horosphere* (in higher dimensions) as the set of normalized *null* vectors with quadratic relation to ∞. These null vectors of the horosphere in $\mathbb{R}^{n+1,1}$ represent *points* in $\mathbb{R}^n$.

This new representation of a *point* is different from a pure Euclidean *vector.* We will see that many other types of geometric elements are naturally encoded in the subspaces of $\mathbb{R}^{4,1}$, including real and imaginary circles and spheres.[23] Table 11 on the next page is a comprehensive list of the most commonly used constructions. Before we explore those elements, however, the following section will focus on the various kinds of *bivectors* available to us in the conformal model.

[22]One might notice that, essentially, as first order expansion of $d\boldsymbol{x}$, our new point measures its relation to the global space by including its own differential. This creates a highly *localized* set of information that is leveraged by operations later on.

[23]Though typically used to project onto a Riemann sphere from the complex plane, or here to project 3D space onto a hypersphere, the conformal mapping is also possible in *any* dimension: for instance, to map 4-dimensional spacetime to 6 dimensions.

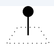
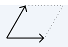
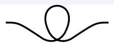
Graphic Symbol	Geometric State	Grade(s)	Algebraic Form	Abbr.
α	Scalar	0	α	`Sca`
	Vector	1	$\boldsymbol{a} = \alpha e_1 + \beta e_2 + \gamma e_3$	`Vec`
	Bivector	2	$\boldsymbol{B} = \boldsymbol{a} \wedge \boldsymbol{b}$	`Biv`
	Trivector	3	$\boldsymbol{I}_3 = \boldsymbol{a} \wedge \boldsymbol{b} \wedge \boldsymbol{c}$	`Tri`
	Point	1	$p = o + \boldsymbol{a} + \frac{1}{2}\boldsymbol{a}^2\infty$	`Pnt`
	Point Pair	2	$\tau = p_a \wedge p_b$	`Par`
	Circle	3	$\kappa = p_a \wedge p_b \wedge p_c$	`Cir`
	Sphere	4	$\Sigma = p_a \wedge p_b \wedge p_c \wedge p_d$	`Sph`
	Flat Point	2	$\Phi = p \wedge \infty$	`Flp`
	Line	3	$\Lambda = p_a \wedge p_b \wedge \infty$	`Lin`
	Dual Line	2	$\lambda = \boldsymbol{B} + \boldsymbol{d}\infty$	`Dll`
	Plane	4	$\Pi = p_a \wedge p_b \wedge p_c \wedge \infty$	`Pln`
	Dual Plane	1	$\pi = \boldsymbol{n} + \delta\infty$	`Dlp`
	Minkowski Plane	2	$E = o \wedge \infty$	`Mnk`
	Direction Vector	2	$\boldsymbol{t}\infty$	`Drv`
	Direction Bivector	3	$\boldsymbol{B}\infty$	`Drb`
	Direction Trivector	4	$\boldsymbol{I_3}\infty$	`Drt`
	Tangent Vector	2	$o\boldsymbol{t}$	`Tnv`
	Tangent Bivector	3	$o\boldsymbol{B}$	`Tnb`
	Tangent Trivector	4	$o\boldsymbol{I}_3$	`Tnt`
	Rotor	0, 2	$\mathscr{R} = e^{-\frac{\theta}{2}\boldsymbol{B}} = cos\frac{\theta}{2} - sin\frac{\theta}{2}\boldsymbol{B}$	`Rot`
	Translator	0, 2	$\mathscr{T} = e^{\frac{\boldsymbol{d}}{2}\infty} = 1 - \frac{\boldsymbol{d}}{2}\infty$	`Trs`
	Motor	0, 2, 4	$\mathscr{M} = e^{\boldsymbol{B}+\boldsymbol{d}\infty}$	`Mot`
	Dilator	0, 2	$\mathscr{D} = e^{\frac{\lambda}{2}E} = cosh\frac{\lambda}{2} + sinh\frac{\lambda}{2}E$	`Dil`
	Boost	0, 2	$\mathscr{B} = e^{o\boldsymbol{t}} = 1 + o\boldsymbol{t}$	`Trv`

Table 11: Basic elements of conformal geometric algebra and their algebraic constructions. The graphic symbols on the left are introduced to help reference the appendix of operations. Bold symbols represent Euclidean elements, with lowercase letters representing 1-blade vectors as is the custom in geometric algebra texts.

4 Groups and Transformations

"Formlessness is proof against the prying of the subtlest spy and the mechanations of the wisest brain."

Sun Tzu

4.1 Subspaces as Tensors

We have seen that the anticommutivity of the wedge product results in an *oriented* algebra. This notion of a *polarity* to the way things combine can also be seen in three grade-dependent properties known as *reversion*, *involution*, and *conjugation*. Tensor analysis includes descriptions for whether matrices are symmetric and hermitian (invariant under reversion) or skew-symmetric and anti-hermitian (antiautomorphic under reversion). Similarly, in geometric algebras we classify how particular grades flip their signs under self-transformations. The Table below reveals how the different grades represent different kinds of tensors. Clifford conjugation can be considered a logical `AND` comparison of reversion and involution. The pattern repeats every 4 grades.

Grade of Blade	Reversion$X \mapsto \tilde{X}$	Involution$X \mapsto \hat{X}$	Conjugation $X \mapsto \bar{X}$
0	+	+	+
1	+	-	-
2	-	+	-
3	-	-	+
4	+	+	+
5	+	-	-

Three simple algorithms can be used to determine this pattern, using k as the grade of the blade.

Reversion $X \mapsto \tilde{X}$:

$$\tilde{X} = -1^{\frac{k*(k-1)}{2}} X \tag{4.1}$$

Involution $X \mapsto \hat{X}$:

$$\hat{X} = -1^{k} X \tag{4.2}$$

Conjugation $X \mapsto \bar{X}$:

$$\bar{X} = -1^{\frac{k*(k+1)}{2}} X \tag{4.3}$$

With the reversion operator we can define the *inversion* of a multivector $X \mapsto X^{-1}$.

$$X^{-1} = \frac{\tilde{X}}{||X\tilde{X}||} \tag{4.4}$$

Since we have defined the product of two geometric entities, as well as their inversion, it becomes possible to speak about the *ratio* of different entities. This is where the expressive power of geometric algebra takes effect, as it gives us a convenient way to *continuously differentiate* between two positions or velocities, and provides a basis for *change*. The ratio of vectors a and b is called an *even versor* or a *rotor* or *spinor* and can also be expressed as the exponentials of some bivector B. *Thus rotors are exponentials of bivectors*, and can be written $\mathscr{R} = e^B$ where e is the canonical exponential (*not* a basis blade) and B is a bivector. Typically, the exponential form expands using simple trigonometric functions (see Table 11). The *rotors* are sandwiched around other elements to transform them: $v^{'} = \mathscr{R} v \mathscr{R}^{-1}$ (sometimes written $v^{'} = \mathscr{R} v \tilde{\mathscr{R}}$ with the tilde representing *reversion).*

We can also compose versors with an odd number of blades (for instance, just one), but such entities do not result in continuously differential (iterative) results. Reflections and inversions are examples of transformations generated by *odd* versors. Rotations and transversions are examples of a transformation generated by *even* versors. This terminology of 'even' and 'odd' comes from Lie algebras. There, one speaks of the *generators* of the algebra, which themselves form a Lie *group*. These groups of generators are classified, as we have been considering, by what kinds of transformations they create in the algebra. For instance, the "pin" group generates reflections and the "spin" group generates rotations. These groups are also classified in terms of what properties they leave invariant after a transformation (e.g. orthogonal groups preserve the inner product). There is a mathematical theory – the Lie-Cartan theory – which states that these groups *completely* define the transformations they describe. Certain Lie groups are isomorphic to geometric algebra, most notably $SO(3)$ (orthogonal rotations) and $SE(3)$ (Euclidean screws). Thus it is in our model that bivectors are the group that completely define the transformations they describe. This is a very powerful concept, as it enables us to linearly combine members of these groups for interesting interpolations.

Lie groups are related to Lie algebras through an *exponential mapping*. Similarly, our algebraic transformations are expressed as exponentials of bivector groups. More specifically, *rotors* (spinors) are exponentials of bivectors, and can be written as $\mathscr{R} = e^B$ where e is the canonical exponential (*not* a basis blade) and B is a bivector.

All continuous transformations – those that are differentiable (e.g. rotations but not reflections) – are generated, in geometric algebra, by bivectors. There is another theorem – the Cartan-Dieudonné theorem – which specifies that the group of generators for orthogonal transformations can be composed of reflections in *well-chosen planes*. Since bivectors represent planes, it makes sense that they should be the basis for all our transformational requirements. As we will see, not all planes in the conformal model $\mathbb{R}^{4,1}$ behave the same way.

4.2 Well-Chosen Planes

Now we consider the new types of *bivectors* that are possible in our 5D model, since these are the elements we will exponentialize to generate *versors* that transform our elements.

The 5D conformal model includes the bivectors of Euclidean 3D space. In Figure 2.2 we show that these Euclidean bivectors represent planes. From Table 6 we know that Euclidean bivectors create a negative term when multiplied together. This negative term tells us that the transformations they enable are *spacelike*.

But in the 5D conformal model there are also other types of bivectors and hence other types of planes. For instance, there is the Minkowski plane $o \wedge \infty$, which squares to a positive term, as well as planes formed between the Euclidean and Null basis, $\boldsymbol{v}\wedge$o and $\boldsymbol{v}\wedge\infty$, which square to 0. These other kinds of planes enable different kinds of transformations – namely *timelike* and *lightlike* depending upon whether they square to a positive term, or to zero, respectively.

Because of the generality in speaking about *spacelike*, *timelike,* and *lightlike* planes, and the fact that these are two-dimensional planes within a higher dimension, we call all of these planes *hyperplanes.* Since they are a vector space they can be added together continuously. Also, composites planes are possible – for instance $\boldsymbol{a}\wedge\boldsymbol{b}+\boldsymbol{v}\wedge\infty$, which is a combination of a rotation plane and translation plane, which creates an interpolatable *dual line* twist axis.

4.3 Reflections:

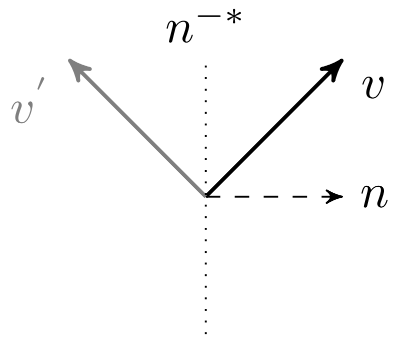


Figure 4.1: $v' = -nvn^{-1}$ defines the reflection of v about the plane with normal versor n. Since n is a unit length 1-blade, $n = n^{-1}$ and the equation for reflection can be simplifed to $v' = -nvn$. The negative sign depends on the dimension of the transformed element: a more general expression is $X' = n\hat{X}n^{-1}$.

Reflections are the base upon which all other transformations operate: all transformations can be considered a combination of reflections. In the conformal model it becomes possible to reflect in a circle or a sphere.

4.4 Rotations:

Euclidean Spin Rotors are generated by a spacelike **Bivector** $B = e_{12} + e_{13} + e_{23}$ with $B^2 < 0$ and weighted bases (i.e. αe_1, βe_2, γe_3). The exponential expression $\mathscr{R} = e^{\boldsymbol{B}}$ with $\boldsymbol{B} = \frac{\boldsymbol{I}\theta}{2}$ admits a familiar expansion: $e^{\boldsymbol{B}} = cos\frac{\theta}{2} - sin\frac{\theta}{2}\boldsymbol{I}$.

4.5 Translations:

Translator rotors are generated by a lightlike **Direction Vector** $d = e_1\infty + e_2\infty + e_3\infty$ with $d^2 = 0$ and weighted bases. They can be considered a double reflection in parallel planes, and can be algorithmically generated as the ratio of two *flat points* (see

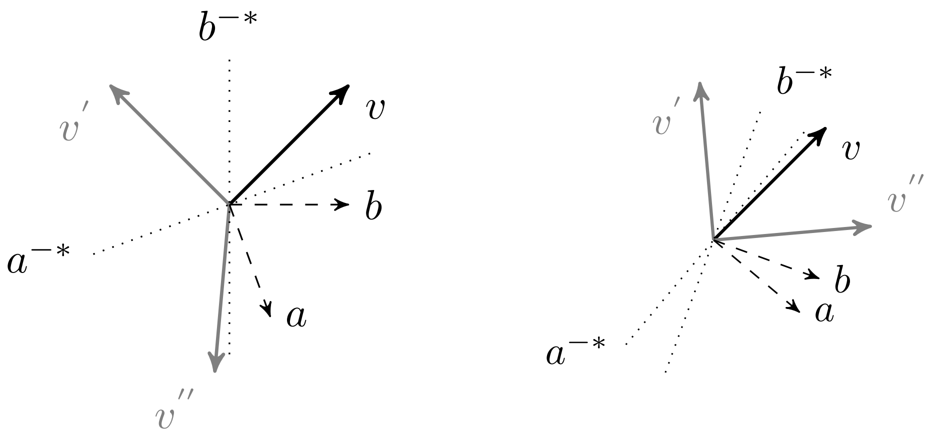


Figure 4.2: $v'' = abvb^{-1}a^{-1}$ defines the reflection of v about two planes with normal versors a and b (a reflection by b followed by a reflection by a). Since a and b are a unit length 1-blades, $a = a^{-1}$ and $b = b^{-1}$. The equation for reflection is thus simplifed to $v'' = abvba$ or $v'' = Rv\tilde{R}$ where ab defines the rotor R, also known as a quaternion or spinor, and $\tilde{R}$ defines the reverse ba. If R is composed of unit versors a and b, then its reverse, $\tilde{R}$, is the same as its inverse R^{-1}. In general you will see this "sandwich" transformation written both ways, as $Rv\tilde{R}$ and/or RvR^{-1}.

Section 5.3.2). We can easily create them according to the algorithm in Table 11. The lightlike exponential

4.6 Dilations:

Dilation rotors about the origin are generated by the timelike **Minkowski Plane** $E = o \wedge \infty$ with $E^2 > 0$ and weighted base. A dilatation around any point in space can be constructed by translating the Dilator $\mathscr{D}$.

4.7 Twists:

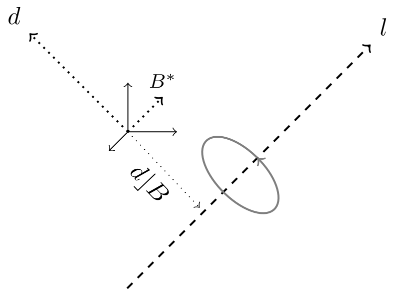


Figure 4.3: A general rotation about the line l with direction a and moment d. $a \cdot d = 0$.

Motors are generated by a spacelike **Dual Line** $l = e_{12}+e_{13}+e_{23}+e_1\infty+e_2\infty+e_3\infty$ (a combination of rotation $B = e_{12} + e_{13} + e_{12}$ and translation $d = e_1\infty + e_2\infty + e_3\infty$

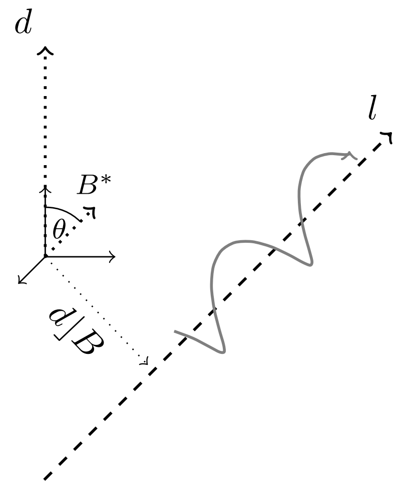


Figure 4.4: A screw rotation about the line l with direction a and moment d. $a \cdot d = \theta$ which defines the *pitch* of the screw along the axis.

bivectors) with $l^2 < 0$ and weighted bases. The six-element bivector generators are isomorphic to the lie group $SE(3)$ and to *Plücker Coordinates.* By exponentiating these elements of the form e^{B+d} we generate an eight-term Motor $\mathcal{M} = s + e_{12} + e_{13} + e_{23} + e_1\infty + e_2\infty + e_3\infty + e_{123}\infty$. Clifford termed his motors *biquaternions* which today are sometimes referred to as *dual quaternions.* These concepts amount to the same thing: general rigid body movements. Chasles theorem provides the foundation for this: any movement in space can be analyzed as a rotation around and translation along some axis in space. This is commonly referred to as *screw theory* and is critical in robot locomotion design.

The dual line generator l consists of a euclidean bivector part B and a direction vector d. The bivector determines the *axis* of rotation, and the direction vector its *moment* from the origin. The angle between B^* and d determines a *pitch* of a screw-like motion, or twist. If B^* and d are orthogonal, the dual line describes a *general rotation* which is a rotation about an arbitrary line in space. Otherwise, if B^* and d are at some angle other than perpendicular, then the dual line l represents an axis of rotation *and* translation. In the case that B^* and d are parallel – that is, identical – the dual line goes through the origin.

Every position and orientation in Euclidean space can be uniquely described by a corresponding twist from the origin. Leo Dorst has demonstrated the ability to combine dual lines in a similar way that one might combine vectors: that is, we can interpolate between them using cubic and quadric bezier techniques[25]. This suggests an immensely powerful and underused approach to designing trajectories – e.g. for navigating immersive environments. In addition, we now have a simple six term description of the twist velocity of an entity. *Versor* includes a class `Frame` that incorporates these principles. The algorithm for creating the exponential of a dual line, and conversely, of finding the logarithm for a motor, are described by Dorst, Mann, and Fontijne in [6], and by Wareham, Cameron and Lasenby in [26].

Sommer, Rosenhahn, and Perwass demonstrated that the twist representation of position and orientation allows for discretizable shape descriptors, and thus "uni-

(a) (b) (c)

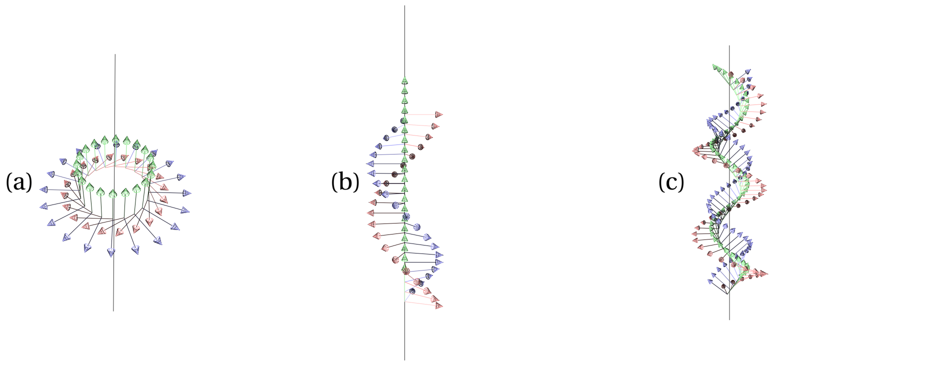


Figure 4.5: Twist Constructions in *Versor* through exponentiation of the dual line axis. (a) General Rotation with orthogonal translation and rotation components: $a \cdot d = 0$. (b) Screw rotation with parallel translation and rotation components $a \cdot d = 1$. (c) extrapolated Screw Rotation.

(a) (b) (c)

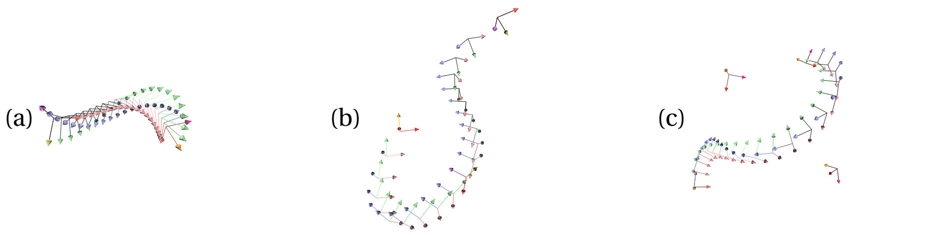


Figure 4.6: (a) Linear (b) Quadric and (c) Cubic interpolations of motor-generating twists.

fies geometry, kinematics, and signal theory"[19]. As has been a central theme in our discussion, Sommer et al. point out that Conformal Geometric Algebra allows for entities and actions to coexist in one framework. Rosenhahn's dissertation, "Pose Estimation Revisited" investigates fourier analysis of twists in order to improve computer vision recognition of arbitrary forms[12].

4.8 Boosts:

Boosts are generated by a lightlike **Tangent** $T = oe_1 + oe_2 + oe_3$ with $T^2 = 0$ and weighted bases. Boosts, also known as *constant accelerations*, *transversors*, or *special conformal transformations*, can play a role in calculating relativity dynamics, as a differential that moves along various orbits. Here we observe how they can be used to *bend* lines into circles, and planes into spheres. Consider the transformations in figure 4.7.

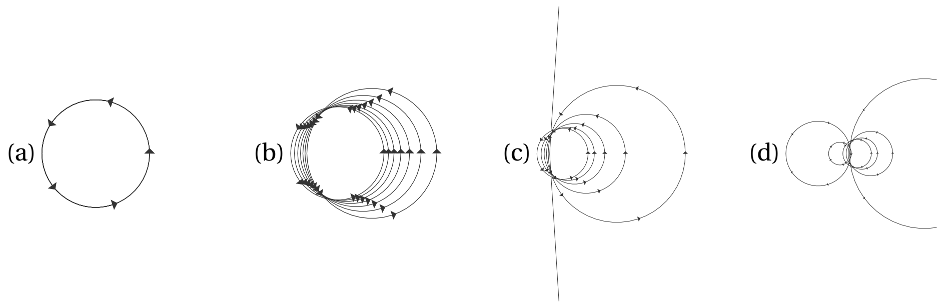


Figure 4.7: $\sigma' = \mathscr{B}\sigma\tilde{\mathscr{B}}$ with σ representing a unit circle (a) at the origin on the e_{12} (xy) plane and $\mathscr{B} = e^{\lambda o \wedge e_1}$ a boost in the e_1 direction. Arrows on the circles show the orientation of the circle. The circle transforms into a line when $\lambda = 1$. When λ is greater than 1, the unit circle has turned inside-out and reverses orientation.

From these an important discovery can be made: the λ in the equation is a convenient representation of the *curvature tensor* of the equation. When $\lambda = 0$ the a unit circle remains unchanged. As λ increases to 1, the unit circle becomes a line. This works in the negative direction as well: as λ decreases to -1, We will see below that such compact canonical representation of the *bend* is a powerful concept in warp computations over a field.

The above depictions follow common textbook depictions of *möbius transformations*. We see the final shape outlined by the transformations *envelope* to be that of a cardioid.

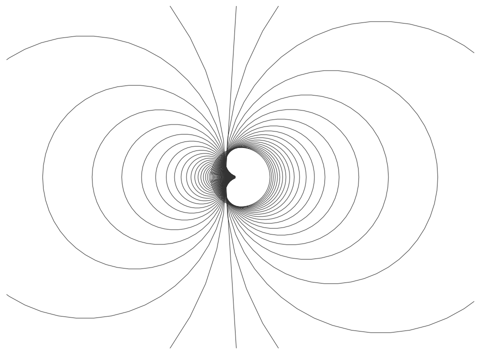


Figure 4.8: Cardioid Envelope of the Special Conformal Transformation operating on a Circle

However, we are no longer limited to the 2D plane. In the above examples, we are using the tangent generator lies in the same plane as the circle it is operating on. What happens if we choose a different tangent or rotate the circle to a different plane? Figure 4.9 below shows the results.

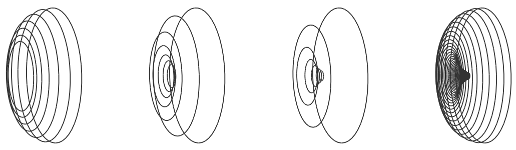


Figure 4.9: $\sigma' = \mathscr{B}\sigma\tilde{\mathscr{B}}$ with σ representing a unit circle (a) at the origin on the e_{23} (yz) plane and $\mathscr{B} = e^{\lambda o \wedge e_1}$ a boost in the e_1 direction.

Another basic form easily created by the transversions are *loxodromes*, which combine the boost tangent with a translation tangent.

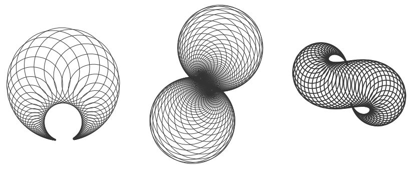


Figure 4.10: Various loxodromic transformations of the form $e^{\lambda o \wedge \boldsymbol{a} + \gamma \boldsymbol{b} \wedge \infty}$.

In the exponent, we can also combine a Euclidean bivector for rotaions such that our versor becomes $e^{\lambda o \boldsymbol{a} + \frac{\theta}{2}\boldsymbol{B}}$, or we can add the Minkowski plane E for dilations. If we combine all these together into one bivector, the resulting exponent is a *point pair:* $\tau = \boldsymbol{B} + o \wedge \boldsymbol{a} + \boldsymbol{t}\infty + E$. Also known as a 0-sphere (a sphere on a line), a point pair is what we get by wedging together to conformally mapped points: $p \wedge q$. As of this writing, there is no algorithm published for calculating a closed form exponential $e^{p \wedge q}$ of a point pair (iterative expansion techniques do exist however) – nor, conversely, is there a logarithm for finding the point pair of such a versor, nor does such a versor have a name. Though nameless, we can tell that the versor created by this exponential must be the ratio of two point pairs which in our program is called a `Mot_Trv` since it is the same as multiplying a motor by a transversor. These compounded transformations contain 16 terms: $\frac{p \wedge q}{r \wedge s} = \alpha + e_{12} + e_{13} + e_{23} + oe_1 + oe_2 + oe_3 + e_1\infty + e_2\infty + e_3\infty + e_{12}E + e_{13}E + e_{23}E + oe_{123} + e_{123}\infty$

Even without a logarithm, we can still use this powerful bivector to construct continuous transformations by assuming we can combine them affinely as can with all the other bivectors. Explorations of such combinations can be seen in Section 5.2.2.

4.8.1 Warped spaces and Curved Trajectories

Conic forms such as ellipses, parabolas, and hyperbolas are not directly represented as subspaces of the algebra[24], but movements along those paths can be easily generated. Here we introduce a method for moving along these curves by specifying a point in space and a tangent vector. If we draw a line connecting the center points

[24]Wareham, Cameron, and Lasenby describe one method in [26], in which a function is fed a vector and an eccentricity. Spinning the vector around carves out various conic shapes.

of the circles of our transformations we can create partial trajectories along elliptical or hyperbolic orbits. Given a round element p with a radius δ and a separate tangent vector $\boldsymbol{ot}$ we can determine a specific position along a hyperbolic or elliptical curve by *boosting* p with the transversor rotor $e^{\boldsymbol{ot}}$. The eccentricity of the curve itself is determined by the relative position and the direction of the tangent vector, and the radius of the original round element. The length of the tangent vector determines the position along that orbit from the particle's current position. The sign of the length indicates the direction of the transformation. Figure 4.11 shows various trajectories carved with a circle and a tangent vector.

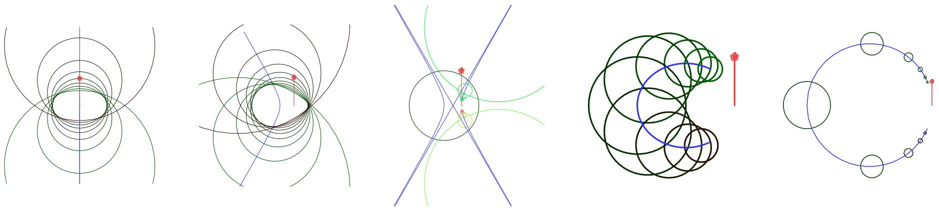


Figure 4.11: As we move a tangent generator (in red) to the right of the circle it operates on, the trajectory of the transformation changes from hyperbolic to elliptic. The switch occurs as the tangent exits the circumference of the circle.

We can generate exotic forms using an interpolated field of tangent vectors to generate boosts: Figure 4.12 depicts the generation of a *warp* field through interpolation of tangent vector generators. Allowing these interpolated boosts to operate on points of a mesh creates interesting dynamic mutations. By treating the field of tangents as direction vectors an *incompressible fluid*, the warp fields observe the confines of a semi-Lagrangian Navier-Stokes solver as presented by J. Stam in [28]. This novel "Hyper Fluid" allows the generation of bounded dynamic deformations, including extrusions/extroversions/invaginations/intussusceptions[25].

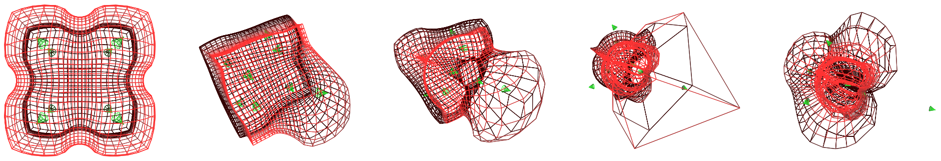


Figure 4.12: Using eight corner tangent vectors to parameterize the hyperbolic warping of a 3D mesh.

It is clear that the rotor form of transformations is quite powerful. [26]

[25]Such shape-shifting is a common characteristic of biological ontogenetic processes.

[26]For instance, a good paper [24] by D. Hestenes and J.Holt clearly translates the international crystallography codes into CGA language using the language of rotors. C. Perwass, the creator of CluViz and CluCalc, has modelled all of the space groups in collaboration with E. Hitzer.

5 Elements

We have described, in Section 3, the quadratic mapping $\boldsymbol{x} \mapsto o + \boldsymbol{x} + \frac{1}{2}\boldsymbol{x}^2\infty$ of a Euclidean vector $\boldsymbol{x}$ to a conformal null-vector representing a homogenous point p. Since we are now dealing in a higher dimension, there are many more ($2^5 = 32$) basis blades with which to work, and the various bivectors, trivectors, and quadvectors that emerge naturally represent basic geometric entities to use in our calculations. We build them up the same way we built up bivector planes in Euclidean space: using the wedge product. Table 11 lists some of the most commonly used entities. Appendix A lists all the basic ways of generating each geometrically significant element in our model: the strict type finite state machine approach used in the implementation of *Versor* enables the easy generation of these useful tables. In this section we look at various representations of these elements – both *dual* and *direct* – and a few of the many ways of constructing them.

5.1 The Meet

Before examining the list of elements in Table 11 it will help to briefly detail the *meet* operation, since the algorithms that define many of the geometric entities are often better understood as intersections of other entities. The dual meet of two entities is constructed by the wedge of the two entities' duals. For instance, in 3D Euclidean space, two bivectors $\boldsymbol{A}$and$\boldsymbol{B}$ are dually defined by their normal vectors $\boldsymbol{A}^*$and $\boldsymbol{B}^*$. The wedge of these two normal vectors $\boldsymbol{A}^* \wedge \boldsymbol{B}^*$ returns another bivector $\boldsymbol{C}$. The dual of $\boldsymbol{C}$ is again a vector $\boldsymbol{C}^*$ which defines the axis of intersection of the original planes $\boldsymbol{A}$ and $\boldsymbol{B}$.

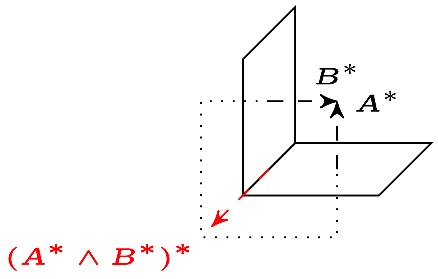


Figure 5.1: Meet of two Blades $\boldsymbol{A}$ and $\boldsymbol{B}$ determined by dualizing their dual wedge: $(\boldsymbol{A}^* \wedge \boldsymbol{B}^*)^*$

We see by this operation that the *meet* is relative to the extension, which is a sort of a *join*.

5.2 Round Elements

5.2.1 Points and Spheres

Normalized homogeneous points, or null-vectors, in the conformal model typically have a weight of 1. They can also be considered as *dual spheres with zero radius*. By adding to or subtracting from the weight of the ∞ basis, we can create *imaginary* or *real* dual spheres of the from $\sigma = p \pm \delta\infty$ where p is the homogenous center point and δ is the radius of the sphere: by adding δ we create imaginary spheres with a negative squared radius. Finding this squared radius is as simple as squaring the dual sphere: $\sigma^2 = r^2$. What exactly an imaginary sphere *is* varies from application to application.[27]

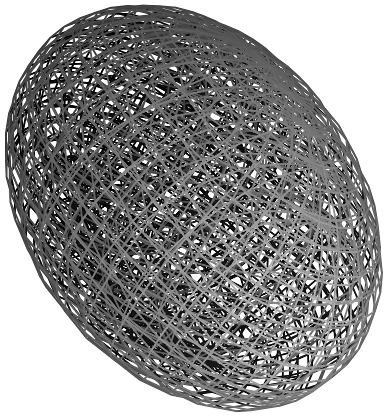


Figure 5.2: Affine combination of two normalized points creates a series of imaginary spheres, the envelope of which resembles an egg.

Null vectors, or points, in the conformal model have the unique property of having a zero dot product with themselves: $p \cdot p = 0$. This interesting result is part of a more general useful trait: the dot product between any two normalized points represents the squared Euclidean distance between them: $p \cdot q = \delta^2$.

The dual of a point is a *direct* sphere: any four points directly define a unique sphere $\Sigma = p \wedge q \wedge r \wedge s$.

5.2.2 Point Pairs and Circles

Wedging two points together forms the ten term bivector $\tau = p \wedge q$. If we consider each point in the pair as a dual sphere, then the operation $p \wedge q$ defines the dual meet of two spheres. If the two unique dual spheres have no radius – that is, if they are in fact points, then their meet is imaginary.

[27] For example: multilayer perceptrons are a tool for statistical analysis of arbitrary sets of data. In [15] Banarer et al examine the imaginary radius of rounds as a confidence measure in determining groupings. While neural nets are beyond the scope of the present paper, it is interesting to point out this particular "role" of the imaginary radius as a measure of certainty since it crops up in experiments. Specifically, it comes into play when determining the meet of a line and a circle: if a line intersects a circle at its perimeter we will find a point. If the line goes through the middle of the circle, we find an inflated point – that is, a point with an extra bit of ∞ in it. As the line moves towards the middle of the circle, the ∞ basis is weighted more heavily until it maxes out when the line goes right through the middle of the circle.

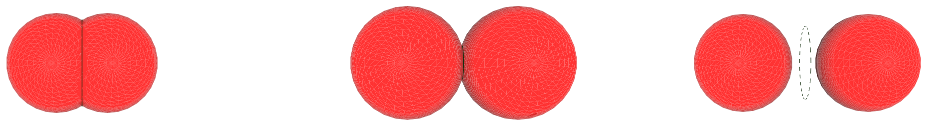


Figure 5.3: Two intersecting dual spheres meet at a circle. As they separate, the circle where they meet shrinks until it becomes a point, and then, when the spheres no longer intersect, becomes imaginary, with a negative squared radius.

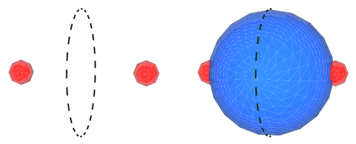


Figure 5.4: Two dual spheres, in red, with a radius close to zero, have an imaginary meet (dotted black circle) and a real "plunge" in blue. The plunge is the surround of the meet, and is orthogonal to both the red spheres that generate it. The term was introduced by Dorst et al in Geometric Algebra for Computer Graphics.

Since point pairs are bivectors, let's assume they can be affinely combined: Using simple linear interpolation techniques, we add two weighted point pairs together. The dual of a point pair is the circle defined by their meet, so we draw this dual representation of our interpolated pairs.

We have seen circles, both imaginary and real, as the dual of point pairs (aka the dual meet of dual spheres). Any three points directly define a unique circle $k = p \wedge q \wedge r$, though they are sometimes more easily found by intersecting two spheres.

5.2.3 Tangents

Part of the reason rounds can be so elegantly combined is that they contain tangent elements. Pure tangents have zero size but a finite weight. They are created by wedging any Euclidean element (vector, bivector, or trivector) with the origin o. We explore uses of tangent vectors as generators at the origin of the form $o\boldsymbol{t}$ in Section 4. Translation of such elements returns an element very similar to a Point Pair. Future work will require more rigorous examination of tangent bivectors, which are closely related to circles, to generate implicit surfaces, and pure tangent trivectors as zero-sized spheres to generate implicit volumes.

5.3 Flat Elements

5.3.1 Directions

Just as *tangents* support *round* elements, so do *directions* support *flat* elements. Directions are made by wedging any Euclidean element (vector, bivector, or trivector)

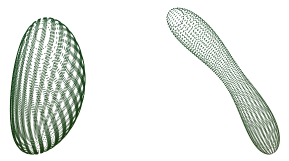


Figure 5.5: Dual representation (as imaginary circles) of affine interpolation of point pairs: $(1-\lambda)p \wedge q + \lambda(r \wedge s)$.

with ∞. Directions are invariant under translations (they do not change if moved), but they can of course be rotated.

5.3.2 Flat Points

Flat points are null vectors wedged with Infinity: $p \wedge \infty$. As Dorst et al explain, they are the result of an intersection between a line and a plane, and they are useful for describing *potential elements* within the algebra. For instance, given a dual line λ and a flat point q not on the line, their union $\lambda \wedge q$ defines a dual plane π through q orthogonal to λ. Similarly, the contraction of a flat point from a direct line $q \rfloor \Lambda$ defines a direct plane Π. Another example: given a dual circle (a point pair) τ and a flat point q, their union $\tau \wedge q$ defines a dual plane that goes through the axis of the circle τ and the point q. We can also construct such a relationship with the contraction product – given a direct circle κ, the contraction with a point $q \rfloor \kappa$ returns a direct plane that goes through the circle κ and the point q.

Figure 5.6: A Circle (in blue) and Line (in green) are contracted with a Flat Point (the small red sphere) to create a red plane orthogonal to the circle and a yellow plane orthogonal to the line.

5.3.3 Lines and Dual Lines

Lines are directly generated by wedging a point pair with ∞, or wedging a point with a *direction* vector. We have examined dual lines closely in our discussion of the *Motor* algebra they generate. There are many ways of finding a dual line; for instance, the central axis l of a circle σcan be found by contraction with infinity: $\infty \rfloor \sigma = l$.

5.3.4 Planes and Dual Planes

Dual Planes $\pi = \boldsymbol{n} + \delta\infty$ are combination of a Euclidean normal vector $\boldsymbol{n}$ plus a weighted infinity ∞ representing the distance from Origin (sometimes called the Hessian distance). Notice the Origin basis o is absent here[28] Given two null points p and q, we can construct the dual plane in between them by simple substraction: $\pi = p - q$: subtracting one normalized point from another eliminates the o blade and returns a vector of the form $\pi = \boldsymbol{n} + \delta\infty$ which represents a dual plane with normal $\boldsymbol{n}$ at distance δ from the origin.

5.4 Infinities

The infinity basis blade ∞ remains invariant under the transformations detailed in section 4. By replacing ∞ with e_- or e_+ in the above formulas, we can construct a model of *spherical* or *hyperbolic* spaces respectively.

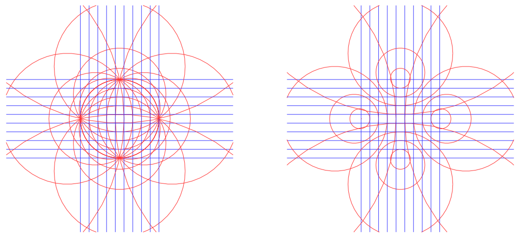


Figure 5.7: The same set of Euclidean lines (in blue) directly defined as $p \wedge \boldsymbol{v} \wedge \infty$ are defined in spherical space as $p \wedge \boldsymbol{v} \wedge e_-$ (left figure) and in hyperbolic space as $p \wedge \boldsymbol{v} \wedge e_+$ (right figure).

Many of these constructions and more can be found in the rich literature on 2D inversive geometry[29] but there is clearly much investigative work that has yet to be done within this recent 3D extrapolation. A direct plane of the form $p \wedge q \wedge r \wedge \infty$ can also be converted into spherical or hyperbolic space by substituting in e_- or e_+ for ∞.

5.5 An Example from Robotics

A good example of the use of many of the above elements exists in the field of robotics, where inverse kinematics strives to define relevant planes and axes of rotation, circles of interest, etc. In a dissertation chapter entitled "Rapid prototyping of robotics algorithms" [27] Dietmar Hildenbrand outlines some of the GA methods used by investigators at CINVESTAV led by Eduardo Bayro-Corrochano. Figure 5.8 recreates the steps required to determine the joint positions in a *kinematic chain* given a desired target.

Such a *k*-chain is representable as a series of *joint motors* specifying the motion at each joint and *link motors* specifying the relative transformation between joints. In the *Versor* implementation, this information is stored as a `Frame`. Sections 6.5.4 and 6.5.3 include a description of classes `Frame` and `Chain`.

[28] In purely affine models, the o element represents the distance, and the ∞ element is absent. See footnote 32 below.

[29] See, for instance, Yaglom's *Complex Numbers* or Needham's *Visual Complex Analysis*.

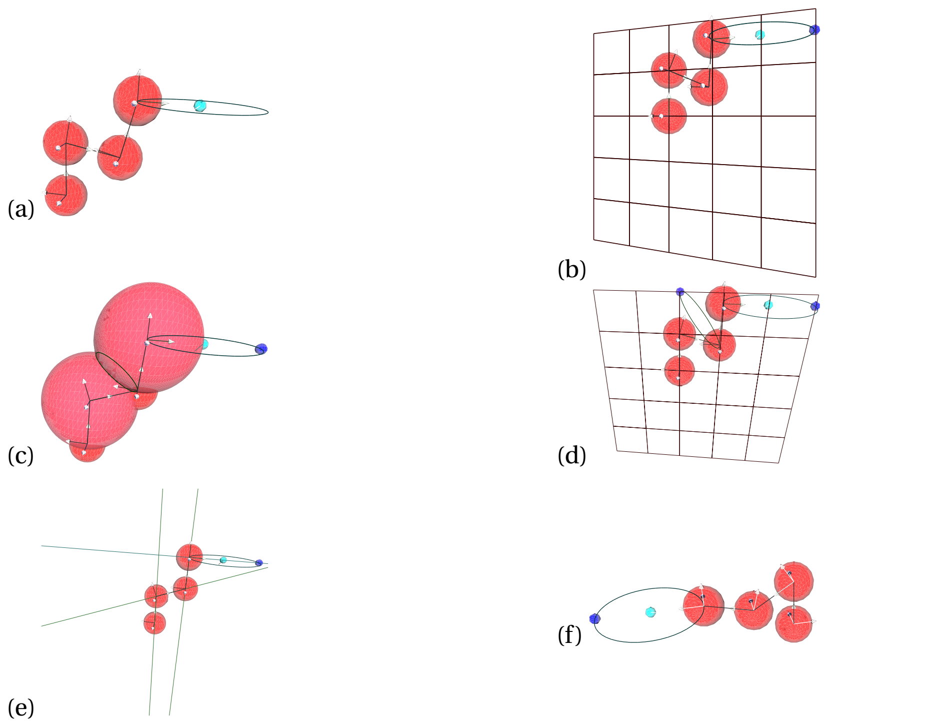


Figure 5.8: A method for calculating the position of an unknown joint in a kinematic chain of joints 0 to 3 entails finding the intersections of spheres and calculating orthogonal planes of interest. (a) The target point defines a circle of interest. (b) The plane through the target point and the robot base intersects with the circle of interest, defining a point pair. The last link on the robot chain, joint 3, is placed at one of these points. (c) Two spheres centered on joints 1 and 3 meet at a circle. This circle defines the set of possible positions for the unknown joint 2. (d) Intersecting the circle with the plane from (b) defines a point pair. The unknown link is set to one of these positions. (e) Orientation of the frames is determined by the dual lines that link them. (f) A different position. For more details the reader is directed to the references [27].

6 Implementation

This section outlines a C++ implementation strategy based on the work of Daniel Fontijne, introduces a unique solution for generating a dynamic library using the Lua programming language, and describes the design of a Finite State Machine for navigating the algebraic operations by treating multivectors as node states and operations as edges for traversing the graph. The construction of a GA-based *Frame* class and a *Field* class is discussed. [30]

6.1 Existing Software

A handful of good stand-alone software packages for visualizing and manipulating CGA currently exist and are familiar to the GA community, specifically *CluCalc* (and *CluViz)*, *Cinderella*, and *GAViewer*. In addition, plugins exist for *Matlab, Maple, Mathematica* and *CGal*, and a fast low-level library *libcga* for C++. There is also an impressive implementation generator called *Gaigen* that can write optimized code in various languages and an optimizer called *Gaalop* that can optimize scripts written for *CluCalc*. Most of these interesting systems are designed as either high-level instructive tools for learning the algebraic logic (typically written in an interpreted language like Java), or conversely as exercises in optimized implementation strategies from a low-level computer science perspective (with little if any built-in physics or application framework). These tools contribute to learning and using the algebra while leaving plenty of room for the creation of more tools: systems that deliver already optimized code within a high-level interface; systems that embed GA within a navigable multimedia environment; systems that incorporate audio to allow for a digital signal processing approach to form synthesis.

6.1.1 Features of *Versor*:

- Immersive Navigation. A class *Frame* simplifies rotations, translations, and twisting –and differentials of each for physics simulations. A subclass *Camera* inherits from Frame greatly faciliating elegant navigation within immersive environments.

- Generator Fields. A templated class *Field* can create a warp field or twist field. This is an extrapolation of classic graphics simulations. Typical 3D direction vector fields for integrating the movment of liquids and fabrics, for instance, can be adapted into Multivector Fields for *twisting* and *boosting* these environments instead, creating *hyper fluids*.

[30] Not detailed here are the libraries with which *Versor* interfaces: the excellent work of my colleagues at the Media Arts and Technology Program in UCSB, who have built lightweight libraries for creating GUIs, adding multimedia device controllers, and interacting with audio sample data. For more information about those, please see the *GLV* library for graphic user interfaces by L. Putnam, G. Wakefield, and E. Newman: `http://mat.ucsb.edu/glv/`, the *Gamma* library for signal processing by Putnam: `http://mat.ucsb.edu/gamma/`, and the OSC messaging manager *DeviceServer* by C. Roberts: `http://www.allosphere.ucsb.edu/DeviceServer/`.

Software	*Interactive*	*Animated*	*Annotated*	*Audio*	*Scripting*	*Integration*	*Optimization*	*Intended Use*	*Unique Feature*
CluViz	yes	25fps	yes (Latex)	–	CluScript	Library or Stand Alone	See Gaalop	Visualization	Annotation
GAViewer	yes	–	yes	–		Stand Alone	–	Visualization	Educational
Cinderella	yes	yes	yes	–	CindyScript	Stand Alone	Jax (for Java)	Education	Physics
Gable	–	–	yes	–	Matlab	Plugin	–	Plotting	
Gaalop	–	–	–	–	CluScript	Optimizer for CluScripts	GPU	Optimization	Speed
Gaigen 2.5	–	–	–	–	C++	Library Generator	Loop Unrolling	Implementation	Generality
MV 1.3.0	–	–	–	–	C++	Library	Sparse Matrices	Implementation	High N-Dim
Versor	yes	yes	–	Gamma	C++	Library or Stand Alone	Loop Unrolling	Immersive Environments	Integration

Table 12: Comparison of Geometric Algebra software.

- Multimodal Interactivity. *Versor* takes advantage of the excellent work done by my colleagues in media infrastructure design, using the gui library *GLV* and the control library *Device Server* to enable interaction with real-time environments. Geometric Algebra can thereby be created and manipulated intuitively through a variety of interfaces, and can be wrapped into larger scale projects.

- Audio Integration. Links to *Gamma* – a cross-platform audio signal processing library (see footnote 30) – enabling experimentation between audio synthesis techniques and 3D form synthesis techniques, as well as hypercomplex signal processing.

6.2 Challenges

The geometric algebraic system poses one main difficult problem for the programmer. To build an efficient implementation one must create various specialized multivector types, which paradoxically preempts the ability to use the generalized nature of the algebra. [31] Loosely following a strategy outlined by Daniel Fontijne, the engineer of *Gaigen* (Geometric Algebra Implementation GENerator), the backbone source code for *Versor* is precomputed by scripts written in *Lua,* allowing for different specialized functions to be called when calculating the various products of each multivector type. Unlike *Gaigen* however, *Versor*'s implementation is locked into the 5D conformal model *(Gaigen* allows users to create any metric in any dimension) and is strictly a C++ program (*Gaigen* allows for implementations in multiple languages). These two limitations grant *Versor* the advantage of running efficiently "off the shelf" – it does not require the user to execute a *Gaigen*-like optimization analysis of his/her own program as part of the design process. Because functions are stored in a dynamic library multiple application(s) can use the *Versor* library at once, each only loading the particular functions it needs into memory.

6.2.1 Goals

A major goal in the design process of this library and software interface was to achieve the following:

- Maintainance of the total general expressivity of the algebra within the conformal model and its homogenous and Euclidean subspaces.

- Optimization by specialization of geometric, exterior, and inner products for each geometrically significant type.

- Implementation of a garden of useful algorithms, including known logarithms for interpolating between general transformations (e.g. twists).

- Ease of compilation and user application design.

[31] In [1] Hildenbrand, Fontijne, Perwass, and Dorst list four difficulties in developing a working GA system: the need for specialized types, the number of basic operations that must be coded, the arbitrary metric, and the exponential increase in combinatoric complexity with each increasing dimension.

In exchange for an optimized self-contained solution, there are limitations:

- Currently only the 5D conformal model is implemented. While this includes efficient implementations of 3D and 4D spaces, it does not allow higher dimensions or alternative metrics. A more flexible approach would allow the implementation of a 6D implementation of Penrose's twistor program, as described in the literature, and other metric spaces which are sure to crop up.
- To ensure the dynamic library component of the software is complete, many functions are included that may likely never be used by any specific application. As a result, relative lightness of the distribution itself is compromised, since it is currently 50mb. Note that, since the library is dynamic, only those functions used are loaded into RAM.
- Since the library is dynamically loaded as function pointers, there is some overhead introduced that could be eliminated with inlined functions.

To solve some of these limitations, a future implementation strategy is proposed in section 6.6, using *JIT* (just-in-time) compilation methods for making the library optimizable, lightweight and capable of handling any metric and any dimension (up to processing power).

6.2.2 Strategy

Even with its current limitations, detailing the method used for this iteration of the project may help others as they learn the algebra, investigate their own implementations, and seek new and innovative formmaking strategies. The process of creating *Versor* can be broken down into the following steps, detailed below.

1. Build an *inefficient* Object-Oriented 3D Euclidean Model. This is adapted from the basis-blade implementation explained in the appendix of *Geometric Algebra for Computer Science.* Basis blades are represented by bitflags as described in Table 3. The geometric, outer, and inner products are calculated with the *xor* operation between basis blades. Multivectors are arrays of basis blades, and are operate on each other as nested for-loops of basis blade-to-basis blade comparisons.
2. Conformalize. Introduce two new bases, one which squares to 1 and another which squares to -1. Create a protocol for mapping between non-degenerate and degenerate metrics as described in equations 3.1 and 3.2.
3. Bake Results. The Object Oriented reference model of steps 1 and 2 are used to generate a Lua Tables Model version. In this model we define each basis blade as a *table* which can index into each other, and each specialized type as a list of these tables. This gives us a succinct method for looking up results of arbitrary operations. We use these tables to create optimized C code in step 4, as well as to examine properties of the algebra – for instance to create the operation tables in the appendix.

4. Generate Library. The Cosmo templating library of Lua is used to write efficient C functions with unrolled loops. Heuristics guide the decisions of which operations should be computed.

6.3 Creating The Object Oriented C++ General Reference Model

6.3.1 Euclidean Geometric Algebra

The Basis Class We first build a bit-representation of basis blades. To make things easier, we use the `bitset<>` class from the C++ standard template library. This class allows for easy summation of the number of "on" bits (which tells us the grade of the blade). For a 3D Euclidean model we define a class `Basis` which has an `mBlade` member of type `bitset<3>` and a `mWeight` member of type `double`.

The class must include methods for involution, reversion, and conjugation as defined in equations 4.1 through 4.3. We also define three operators, `*`, `<=`, and `^` to represent the geometric, inner, and outer products between basis blades, respectively. All products are functions of two basis blades which output an `xor` comparison of the operands. The inner and outer products are treated as special cases of the geometric product, and return an empty Basis if certain conditions are not met.

Algorithm 1 Geometric Product of Basis Blades

```
function PRODUCT(A, B)
    $r \leftarrow A.blade \oplus B.blade$                ▷ $\oplus$: bitwise XOR
    $w \leftarrow A.weight * B.weight$
    $s \leftarrow order(A.blade, B.blade)$               ▷ Check for sign flip
    return $Basis(r, w * s)$
```

The `order` function in the the above algorithm is critical for anticommutivity: it checks the relative positions of the bits and determines whether the result needs to be multiplied by `-1`.

Algorithm 2 Determine the order of blades and return 1 or -1

```
function ORDER(a, b)
    repeat
        $a >>= 1$                                     ▷ bitshift right
        $c \leftarrow a \& b$                          ▷ bitwise AND
        $n \leftarrow n + c.count()$                   ▷ accumulate result
    until $a = 0$
    if $n \& 1$ then
        return $-1$
    else
        return $1$
```

Calculating the inner product starts by calculating the geometric product, with two conditionals. Here we are only interested in the Hestenes left contraction product, which ensures the left operand is of a lower grade than the right operand, and also discards results that are not specifically of grade $B_{grade} - A_{grade}$. Alternatively contraction products are easily made with different conditionals (to allow right hand contractions for instance).

Algorithm 3 Inner (Left Contraction) Product of Two Basis Blades

```
function INNER(A,B)
    $r \leftarrow product(A,B)$
    if $A.grade > B.grade$ or $r.grade \neq (B.grade - A.grade)$ then
        return $Basis()$                                  ▷ return an empty Basis
    else
        return $r$
```

The outer product simply checks for any common blades, and discards those results.

Algorithm 4 Outer Product of Two Basis Blades

```
function OUTER(A,B)
    if $A.grade \& B.grade \neq 0$ then                   ▷ check for any shared bits
        return $Basis()$                                  ▷ return an empty basis
    else
        return $product(A,B)$
```

That covers the basics, though it will likely be useful to define a comparison operator between basis elements to be used in the Multivector class described below.

The Multivector Class Multivectors are treated as arrays of Bases. Calculating the geometric, inner, and outer products of multivectors A and B is a linear operation, and therefore just a matter of iterating through each Basis member of A and calculating the product with each Basis member of B.

Algorithm 5 Geometric Product of Multivectors

```
function PRODUCT(A,B)
    for $i$ in $A.bases$ do
        for $k$ in $B.bases$ do
            $r \leftarrow product(i,k)$
            $c \leftarrow c + r$      ▷ $r$ is added to the list of basis blades contained in c
    $c.compress()$                                        ▷ combine like terms
    $c.clean()$                                  ▷ eliminate 0 terms return $c$
```

The `compress()` and `clean()` methods in the above pseudocode keep the resulting Multivector as sparse as possible.

6.3.2 Conformal Geometric Algebra

Introducing a new signature to the underlying vector space is straight forward, though creating a null basis complicates matters slightly. Continuing to follow the implementation guidelines established by Dorst, Mann, and Fontijne, we first need a new function which determines whether the blades flip signs under the metric. This information is stored as an array filled with values of 1 or -1. Our $\mathbb{R}^{4,1}$ metric, for example, is represented as an array of doubles:

```
double Signature::R41[5] = {1.,1.,1.,1.,-1.};
```

We define a new metric product function for our `Basis` class which checks for *annihilating* bits – that is, those shared elements of two basis blades. Since the outer product discards operations between basis with similar components, the following algorithm need only be applied to the geometric and inner products.

Algorithm 6 Geometric Product of Basis Blades in $\mathbb{R}^{4,1}$

```
function METRICPRODUCT(A, B, R41)
    $r \leftarrow product(A,B)$                ▷ normal product of algorithm
    $t \leftarrow A\&B$                        ▷ bitwise AND comparison returns annihilating bits
    $i \leftarrow 0$
    while $t \neq 0$ do
        if $t\&1 \neq 0$ then
            $r \leftarrow r * R41[i]$          ▷ Multiply annihilating blades by metric
        $t \leftarrow t >>= 1$                 ▷ bitshift right
        $i++$
    return $r$
```

Finally, we need a way to add in our *null basis,* defined by the relations in equations 3.1 and 3.2. We will be defining our multivectors in terms of these elements, which define the *degenerate* (non-diagonalized) metric of Table 9. The strategy outlined in Algorithm 6 only applies to *non-degenerate* (diagonalized) metric tables, so we will have to create methods for switching back and forth. It is easy to see why this model can be inefficient to compute, yet we will eventually use the results of this section to bake a more efficient model. Still, to avoid drastically increasing our compute time in the reference implementation, we store our metric transformations in a table. BUT it is important that in switching from our degenerate metric into a non-degenerate one, we will be exchanging one basis element for two, and so our look-up table needs to be a table of Multivectors, as opposed to a table of Basis blades.

To build our tables we define two functions: `pushBasis` and `popBasis`. Pushing takes a blade from our null basis and returns a multivector from the diagonalized metric. Popping takes a blade from the diagonalized metric and returns a multivector in the null basis.

In performing bitwise calculations, enumerations are shared between nondegenerate and degenerate metric spaces. These are listed in Table 13.

Minkowski (nondegenerate)	bits	null (degenerate)
EPLANE	11000	ORIINF
EPLUS	01000	ORIGIN
EMINUS	10000	INFINITY

Table 13: Enumerated Bit Representations in the Conformal Model

Algorithm 7 Switching from a null basis to the standard diagonal $\mathbb{R}^{4,1}$

```
function PUSHBASIS(a)
    Multivector $mv$
    $t \leftarrow a.blade$                         ▷ Get the bit representation of Basis $a$
    $w \leftarrow a.weight$                        ▷ Get the weight of Basis $a$
    if $t\&ORIINF == 0$ then                       ▷ compare bits to $E$ plane
        return $Multvector(a)$                     ▷ if no bits shared, return input
    if $t\&ORIINF == ORIINF$ then
        return $Multivector(a)$                    ▷ blade contains $E$ plane, return input
    if $t\&ORIGIN \neq 0$ then
        $b \leftarrow t \oplus ORIGIN$             ▷ remove $o$ bit with $xor$ operation
        $b1 \leftarrow b \oplus EPLUS$             ▷ add in $e_+$ bit
        $b2 \leftarrow b \oplus EMINUS$            ▷ add in $e_-$ bit
        $nb1 \leftarrow Basis(b1, w * .5)$         ▷ multiply original weight by .5
        $nb2 \leftarrow Basis(b2, w * .5)$
        $mv \leftarrow mv + nb1$                   ▷ add the two new bases to the Multivector
        $mv \leftarrow mv + nb2$
        return $mv$
    if $t\&INFINITY \neq 0$ then
        $b \leftarrow t \oplus INFINITY$           ▷ remove $\infty$ bit with $xor$ operation
        $b1 \leftarrow b \oplus EPLUS$             ▷ add in $e_+$ bit
        $b2 \leftarrow b \oplus EMINUS$            ▷ add in $e_-$ bit
        $nb1 \leftarrow Basis(b1, w * -1)$         ▷ multiply original weight by $-1$
        $nb2 \leftarrow Basis(b2, w)$              ▷ multiply original weight by 1
        $mv \leftarrow mv + nb1$                   ▷ add the two new bases to the Multivector
        $mv \leftarrow mv + nb2$
        return $mv$
```

The above algorithm 7 satisfies the defining relations of equation 3.1: $o = .5(e_+ + e_-)$ and $\infty = e_- - e_+$. The next algorithm will satisfy the defining relations of equation 3.2: $e_+ = o - .5\infty$ and $e_- = o + .5\infty$

Algorithm 8 Switching from the standard diagonal $\mathbb{R}^{4,1}$ to the null basis

```
function POPBASIS(a)
    Multivector $mv$
    $t \leftarrow a.blade$                         ▷ Get the bit representation of Basis $a$
    $w \leftarrow a.weight$                        ▷ Get the weight of Basis $a$
    if $t\&EPLANE == 0$ then                       ▷ compare bits to $E$ plane
        return $Multvector(a)$                     ▷ if no bits shared, return input
    if $t\&EPLANE == EPLANE$ then
        return $Multivector(a)$                    ▷ blade contains $E$ plane, return input
    if $t\&EPLUS \neq 0$ then
        $b \leftarrow t \oplus EPLUS$              ▷ remove $e_+$ bit with $xor$ operation
        $b1 \leftarrow b \oplus ORIGIN$            ▷ add in $o$ bit
        $b2 \leftarrow b \oplus INFINITY$          ▷ add in $\infty$ bit
        $nb1 \leftarrow Basis(b1, w)$
        $nb2 \leftarrow Basis(b2, w * -.5)$        ▷ multiply original weight by $-.5$
        $mv \leftarrow mv + nb1$                   ▷ add the two new bases to the Multivector
        $mv \leftarrow mv + nb2$
        return $mv$
    if $t\&EMINUS \neq 0$ then
        $b \leftarrow t \oplus EMINUS$             ▷ remove $e_-$ bit with $xor$ operation
        $b1 \leftarrow b \oplus ORIGIN$            ▷ add in $o$ bit
        $b2 \leftarrow b \oplus INFINITY$          ▷ add in $\infty$ bit
        $nb1 \leftarrow Basis(b1, w)$
        $nb2 \leftarrow Basis(b2, w * .5)$         ▷ multiply original weight by $.5$
        $mv \leftarrow mv + nb1$                   ▷ add the two new bases to the Multivector
        $mv \leftarrow mv + nb2$
        return $mv$
```

6.4 Lua Code Generation

Lua tables can be used as mixed-dimensional multivectors to generate a strictly typed and stripped-down "brute-force" C dynamic library for CGA. The library unrolls loops to precompute the combinatoric operations necessary for efficient CPU computing and stores the resulting functions in a table. This "hard-wires" all of the calculations built by the reference model.

At the core of the Lua language is the notion of a *table* of key-value pairs. These tables behave like a combination of maps and classes in C++ code, with both the *class-like* flexibility of containing mixed types and the *map-like* flexibility of having key-value pairs added after initialization of the table. Additionally, Lua allows one table to use another table as a key or a value. This greatly simplifies our task of organizing the results of the C++ reference model. The table-as-object approach used in Lua conveniently encapsulates some core GA concepts: since multivectors are "objects" with "tabled" rules describing how they combine, Lua is a clean way to program these two critical characteristics of the algebra.

Key	*Value*
`id`	string::name
`w`	int::weight
`g`	int::grade
`gp`	{}
`op`	{}
`ip`	{}

Table 14: Initial Key-Value Pairs for Basis Blade Tables

6.4.1 Building Basis Tables

We use the inefficient iterative `xor` bit logic of our reference construction to generate a combinatoric map of our elements. To start out, all basis blades are defined as individual tables with:

- an id name (e.g. `id="_e1"`)
- a weight of 1 (`w=1`)
- a grade variable (e.g.`g=1`)
- an empty geometric product table (`gp=`)
- an empty outer product table (`op=`)
- an empty inner product table (`ip=`)

In 5D Geometric Algebra, there are 2^5 – or 32 – different basis elements, and so we make a table for each. On initialization, each basis table has empty geometric, outer, and inner product tables. For each one of these empty tables, we will use the basis tables themselves as the "keys" and the "values" will be a new table. Thus each `xor` function from our reference model is converted into a key-value pair. The values themselves will be *one or more* basis elements, with weights of 1, -1, or 0. Since all basis blades are assigned a weight of 1, we also define a simple function `blade(b,w)` which will return a copy of the same basis blade `b` but with a different weight `w`. The original basis blade remains unchanged, with a weight of 1.

For example, our first basis table `_e1` is created:

```
_e1 = { id = "_e1", w = 1, g = 1, gp = {}, ip = {}, op = {}}
```

and all other basis tables are similarily created, with "`no`" and "`ni`" standing for "null origin" and "null infinity" after the custom introduced by Fontijne.

```
_e2 = { id = "_e2", w = 1, g = 1,, gp = {}, ip = {}, op = {}}
...
```

```
_ni = { id = "_ni", w = 1, g = 1, gp = {}, ip = {}, op = {}}
...
_e123noni = { id = "_e123noni", w = 1, g = 5, gp = {}, ip = {}, op = {}}
```

Next, we fill in the empty product tables for each basis element:

```
_e1.gp[_s] = { blade(_e1,1), }
_e1.op[_s] = { blade(_e1,1), }
_e1.ip[_s] = { blade(_s,0) }
_e1.gp[_e1] = { blade(_s,1), }
_e1.op[_e1] = { blade(_s,0) }
_e1.ip[_e1] = { blade(_s,1), }
_e1.gp[_e2] = { blade(_e12,1), }
_e1.op[_e2] = { blade(_e12,1), }
_e1.ip[_e2] = { blade(_s,0) }
...
_e1.gp[_e123noni] = { blade(_e23noni,1), }
_e1.op[_e123noni] = { blade(_s,0) }
_e1.ip[_e123noni] = { blade(_e23noni,1), }
```

This would take a long time to do by hand, so luckily we are generating these tables from our reference implementation. It is useful to point out here that each product "key" returns a *table of tables* because there are some products that return more than one basis element – for example, the geomeric product between the origin o and infinity ∞ returns both a scalar (with a weight of -1) *and* a bivector (the Minkowski plane E): $o\infty = -1 + o \wedge \infty$

```
_no.gp[_ni] = { blade(_s,-1), blade(_noni,1), }
```

Lastly, each basis table must return values for `involution`, `reversion`, and `conjugation` keys which specify whether or not there is a sign flip. Since these operations only change the sign of a blade, and don't create new blades, we do not need to worry about the possibility of multiple outputs. For instance, the _e12 basis blade:

```
_e12.involution = blade(_e12,1)
_e12.reversion = blade(_e12,-1)
_e12.conjugation = blade(_e12,-1)
```

6.4.2 Building Basic Types

With a table for each 32 basis elements, and geometric, outer, and inner products operations for each, along with involution, reversion, and conjugation operations, the next step is to build up basic types. Heuristically, we know what these should be. As shorthand, we assign all our basic elements three letter nicknames. For instance, our 3D Euclidean subspace vector is a `Vec`, and our 3D Euclidean subspace bivector is a `Biv`. A list of names is in Table 11.

```
Vec = { id = "Vec", desc = "Euclidean Vector", bases = {_e1, _e2, _e3, } }
Biv = { id = "Biv", desc = "Bivector", bases = { _e12, _e13, _e23} }
```

```
    Rot = { id = "Rot", desc = "Rotor", bases = { _s, _e12, _e13, _e23} }
    Mot = { id = "Mot", desc = "Motor", bases = { _s, _e12, _e13, _e23, _e1ni, _e2ni,
_e3ni, _e123ni} }
```

We give our specialized types an id name, a description, a key, and a table of bases (and each basis is itself a table). The `desc` key will come in useful when we print out analyses of the algebra itself. When "choosing" which types to represent, it is important to remember that there are many – more even than are listed in the authoritative book on programming GA by Dorst, Mann, and Fontijne, on the subject. For instance, we have not discussed *Paravectors* which are composed of vectors plus scalars.[32]

The next step is a matter of taking products between these types. We do this the same way we did it in the reference model – as a linear operation where we we calculate the result of performing the operation on each basis element of the first type with each basis element of the second type. We reduce terms by combining similar blades, and eliminate the results of operations which return a blade with a weight of 0. We compare this final, reduced term to our list of pre-built types. If it already exists, we do nothing, if not, we create a new type with a name that reflects the operation used to create it. For instance, the geometric product of a Rotor and Vector is something we have not discussed, and yet it crops up frequently in calculations as an "intermediate" or "temporary" term when performing transformations. We don't have a description for it, so it just uses its id name.

```
    Rot_Vec = { id = "Rot_Vec", desc = "Rot_Vec", bases = {_e1, _e2, _e3, _e123, }
}
```

A transformation of the form $R\boldsymbol{v}\tilde{R}$ actually creates this strange "dual rotor" though such a multivector is rarely mentioned in the literature. The transformed vector $\boldsymbol{v}'$ is simply assigned the first three bases; the last basis, the Euclidean pseudoscalar, is dropped. Such *intermediate* multivectors – those without obvious geometrical representation that still "crop up" during basic calculations – will be included in the finite state machine. Determining which ones are necessary is a heuristic matter; during the course of writing *Versor* it was sometimes necessary to go back to the code generation and include an intermediate type to avoid errors or improve efficiency.

6.4.3 Building a Finite State Machine

From the above tables of basic elements and "intermediate" multivectors we generate a total of 167 types[33] The next step is one of iteration and C code generation: for

[32]Nor have we precisely articulated how the *affine* model fits within the conformal model. Briefly: we can construct Affine points, lines, and planes without the ∞ component. Affine points contain: e_1, e_2, e_3, o. Affine lines contain: e_{12}, e_{13},e_{23},$e_1 o$,$e_2 o$, $e_3 0$. Affine planes contain: $e_{12}o$,$e_{13}o$, $e_{23}o$,e_{123}. See also Bayro-Corrochano, Reyes-Lozano, and Zamora-Esquivel's 2006 text, p.67.[4]

[33]This total includes affine representations of space, as well as a few paravectors, though I have not yet explored such concepts explored in any significant way. See previous footnote.

each of the basic types, we calculate and unroll three operations (inner, outer, and geometric) with each other type, for a total of 83,667 functions and 80mb worth of information (56mb when compiled). Many of these functions will never be used by a given program, so the resultant *libconga* is built as a dynamic library to which *Versor* links. Only those functions used by any given program are loaded into memory. This solution is fast enough, thorough, and self-contained: once it is built, *libconga* can handle practically any operation in our arsenal. In addition, it can easily be ammended with additional types as the need might arise, or fewer, and recompiled. Its weaknesses are the redundancy and size of the dynamic library, and the overhead created by treating all operations as function pointers instead of allowing the compiler to inline them. An alternative solution could be to create program specific libraries (*Gaigen's* approach), or to JIT compile just the necessary functions at runtime. For experimentation and fast prototyping, the finite state machine approach works well.

Each function takes three pointers to doubles representing a left operand, a right operand, and a return. All functions are indexed in an array of function pointers, and an array of integers stores an index to this index, specifying a return type. Each type has three arrays, one for each operation, which index into the function pointer array. When *Versor* executes `Vec a * Rot b`, it looks at the geometric product array in *libconga* for type `Vec` (a 3D Euclidean Vector) and returns an index of the function for multiplying it by a `Rot` (a Rotor). This function is passed in the values of a and b and an empty array to fill and return. *Versor* also looks up what type is returned.

The model described above defines a finite state machine that executes a function from a list of functions, stores the resultant data, and moves to the next list of functions specific to that data type. Because of the precomputed types (Versor "knows" for instance that contracting *infinity* out of a *circle* results in a *dual line* which represents the axis through the circle), it can also be used to answer queries like "given a circle and a point, what can I produce?"

6.5 Adding Functionality

The list of basic operations in Table 15 can be used to construct more complex functions. Working on top of *libconga*, *Versor* implements many of the known algorithms for calculating spatial relationships and transformations, from finding the tangent to a sphere at a specific point to extracting logarithms of Motors. Table 16 below lists some of the more common functions used in *Versor*.[34] These functions are static and blind to the input types: they take in one or more multivectors and spit out another. Some internally force the return of specific type, but inputs and outputs are generic States. The generic operations enable a data flow model for building abstract operation graphs with functionoids: each multivector `State` can be a drawable `Observer` to the `Graph` of operations in which it finds itself.

[34]A complete documentation has been generated with Doxygen and is available online at `www.mat.ucsb.edu/versor/`

Operation	Operator	Expression	Notation
Geometric Product	`*`	`A * B`	AB
Inner Product	`<=`	`A <= B`	$A\rfloor B$
Outer Product	`^`	`A ^ B`	$A \wedge B$
Commutator	`%`	`A % B`	$A \times B$
Inverse	`!`	`!A`	A^{-1}
Reverse	`~`	`~A`	$\tilde{A}$
Conjugate	`conjugate()`	`A.conjugate()`	$\bar{A}$
Involute	`involute()`	`A.involute()`	$\hat{A}$

Table 15: How to perform basic operations on multivectors in *Versor*

Function	Output	Notation
`Op::sp`	Spin A by B	$\mathscr{B}A\mathscr{B}^{-1}$
`Op::re`	Reflect A by B	$\mathscr{B}\hat{A}\mathscr{B}^{-1}$
`Op::rj`	Rejection of A from B	$(A \wedge B)B^{-1}$
`Op::pj`	Projection of A onto B	$(A\rfloor B)B^{-1}$
`Op::dl`	The Dual of A	A^{*}
`Op::ud`	The Undual of A	A^{-*}
`Op::dle`	The Euclidean Dual of A	$A^{\star}$
`Op::ude`	The Euclidean Undual of A	$A^{-\star}$
`Op::mat`	The 4x4 matrix of input Rotor $\mathscr{R}$	$\begin{bmatrix} x_0 & y_0 & z_0 & 0 \\ x_1 & y_1 & z_1 & 0 \\ x_2 & y_2 & z_2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
`Op::aa`	The axis angle rep of input Rotor $\mathscr{R}$	$\begin{bmatrix} \theta & x & y & z \end{bmatrix}$

Table 16: Some useful functions operating on one or two arguments.

Function	Output	Notation
`Gen::log_rot`	The Bivector Generator of input $\mathscr{R}$	$log(\mathscr{R})$
`Gen::log_mot`	The Dual Line Generator of input$\mathscr{M}$	$log(\mathscr{M})$
`Gen::mot_dll`	The exponential $\mathscr{M}$ of input Dual Line $\boldsymbol{B} + \boldsymbol{d}\infty$	$e^{\boldsymbol{B}+\boldsymbol{d}\infty}$
`Gen::rot_biv`	The exponential $\mathscr{R}$ of input Bivector $\theta\boldsymbol{I}$	$e^{-\frac{\theta}{2}\boldsymbol{I}}$
`Gen:ratio_vec`	The Rotor $\mathscr{R}$ that takes input Vec $\boldsymbol{a}$ to input Vec $\boldsymbol{b}$	$\frac{(1+\boldsymbol{ba})}{\sqrt{2(1+a\cdot b)}}$
`Gen::ratio_dll`	The Motor $\mathscr{M}$ that takes input Dll A to input Dll B	
`Gen::trs`	The exponential $\mathscr{T}$ of input Direction Vector $\boldsymbol{d}\infty$	$e^{-\frac{\boldsymbol{d}}{2}\infty}$
`Gen::dil`	The exponential $\mathscr{D}$ of input E plane λE	$e^{\frac{\lambda}{2}E}$
`Gen::trv`	The exponential $\mathscr{B}$ of input tangent vector	$e^{\frac{o\boldsymbol{t}}{2}}$

Table 17: Common functions for dealing with versors and their generators.

Function	Output
`Ro::null`	Point p map of input Vector $\boldsymbol{v}$
`Ro::dls`	Dual Sphere from input Vector $\boldsymbol{v}$ and radius α
`Ro::split1`	Point p of input Point Pair $p \wedge q$
`Ro::split2`	Point q of input Point Pair $p \wedge q$
`Ro::sur`	Dual Sphere Surrounding input
`Ro::cen`	Center point p of input Round
`Ro::car`	Carrier Plane or Line of Input Circle or Point Pair
`Ro::siz`	Squared Radius (+ or -) of input Round
`Ro::wt`	Weight α of input Round

Table 18: Common functions for creating and querying round elements

Function	Output
`Fl::car`	Carrier Plane or Line of Input Circle or Point Pair
`Fl::loc`	Point p on input line or plane closest to input point
`Fl::wt`	Weight α of input Flat
`Fl::dir`	Direction of input Flat

Table 19: Common functions for creating and querying flat elements

Function	Output
`Ta::at`	Tangent to input State at input point p
`Ta::wt`	Weight α of input Tangent

Table 20: Common functions for creating and querying tangent elements

Dorst, Mann, and Fontijne's textbook includes a table of useful operations for extracting the position, orientation, and size of various elements of the algebra. Some are trivial (the direction of a tangent are the same three coordinates) and not implemented algorithmically. To calculate draw routines this information is translated into OpenGL matrix calls, `glTranslate*`, `glRotate*`, and `glScale*`. For instance, here is the `draw()` routine called by a `Cir` (circle) element `myCircle`.

```
//Center:
   Pnt v = Ro::cen( myCircle );
//Orientation Relative to xy plane:
   bool sign = Op::sn(b, Biv::xy);
//Unit Euclidean Bivector Plane of Circle:
   Biv B = Biv( Ro::dir( myCircle ) ).unit();
//Rotor to B:
   Rot r = Gen::ratio_vec( Vec::z, Op::dle( B ) );
//Axis angle of Rotor:
   Vec4 t = Op::aa(r);

//Squared Radius:
   double siz = Ro::siz( myCircle );
//Is it an imaginary Circle?
   bool im = siz > 0 ?  1 :  0;
//Radius:
   double rad = Ro::rad( myCircle );
//Transform OpenGL Model View Matrix According to Center and Orientation:
   glTranslated(v[0],v[1],v[2]);
   glRotated(t.w, t.x, t.y, t.z);
//Draw Real or Imaginary Clockwise or Counterclockwise Circle:
   im ?  Glyph::DirCircle(rad,sn) :  Glyph::DirDashedCircle(rad,sn);
```

6.5.1 Interpolations and Filtering

Linear, quadric, and cubic interpolations of multivectors are possible by feeding an array of them to the `Interp::Linear|Quadric|Cubic` function, and can be a closed loop or open. These simple interpolating functions are leveraged by the Field class, described below. Additionally, the *Gamma* signal processing library is generic enough that it can be used to filter a set of multivectors, since we have suitably overloaded the necessary functions.

6.5.2 The Frame Class

As an example of how to use the algebra in physics simulations and camera navigation, let us consider a localized class `Frame` which can move, rotate, screw, dilate, or boost about in a space. A `Frame` will need an orientation and a position, which will be stored as a *Rotor* and a *Point* in member variables `mRot` and `mPos`. It will also store differentials for each of the transformations in the form of small bivectors `dDrv`, `dBiv`, `dDll`, `dMnk`, and `dTnv` (A *Direction Vector*, a Euclidean *Bivector*, a *Dual Line*, a *Minkowski Plane*, and a *Tangent Vector*, respectively). Finally, it will store acceleration values for each of these differential operators in the form of scalars `aDrv`, `aBiv`, `aDll`, `aMnk`, and `aTnv`.

At each time step of our simulation, `Frame` calls a `step()` method which multiplies all the bivector differentials by their respective acceleration coefficients, and then uses the bivectors to generate a Translator, Rotor, Motor, Dilator, and Transversor. These versors are applied to the `mRot` and `mPos` variables of `Frame` in methods called `move()`, `spin()`, `twist()`, `scale()`, and `boost()`.

A `Frame` has three methods, `right()`, `up()`, and `forward()`, which return the local axes x, y, and z as vectors, direction vectors, or tangent vectors. This local coordinate system is stored in a 4x4 matrix `mImage` made by rotating the global x, y, and z axes by `mRot` using the `Op::mat` function from Table 16. This matrix, or *image*, is updated every step through a call to `Op::mat(mRot)` within an `orient()` method.

`Frame` also has a `mot()` method which returns a *normalized motor* created by multiplying its translator – `Gen::trs(mPos)` – by its rotor `mRot` and normalizing the result with `runit()`. This motor uniquely describes a position and orientation in space relative to the origin. By taking the log of this normalized motor through the `Gen::log_mot` method of Table 17, we get an interpolatable dual line which we can use to twist one reference frame to another. This is the method used to generate Figure 4.6.

6.5.3 The Chain Class

The `Frame` class described above is used to build a class `Chain`, with members `mJoint` and `mLink`, both arrays of Frames of size `mNum`. Each joint frame i is paired with a link frame describing the relative position of the next joint frame $i+1$. Forward kinematics are determined at each joint by concatenating the current motor with the previous link motor and the total transformation of the previous joint.

```
void fk() {
     for (int i = 1; i < mNum; ++i){
         Mot m = mJoint[i].mot() * mLink[i-1].mot() * mFrame[i-1].mot();
         mFrame[i].mot( m );
     }
}
```

6.5.4 The Field Class

The `Field` class provides a framework for 3D Lagrangian, semi-Lagrangian, and Eulerian integration techniques acting on a voxel grid of arbitrary data types.

The most common types of *vector fields* implemented in computer graphics are really *direction vector fields*. `Field` is a templated class which can be composed of any kind of multivector, direction vector or otherwise. With methods for diffusion, advection, and maintaining incompressibility (based on J. Stam's semi-Lagrangian technique outlined in [28]), it creates dynamic fields of *interpolatable and integratable bivector generators* in 1, 2 or 3 dimensions. These generators are used to create a perturbation variance which can be applied to a `Field` of any other type.

The images in Figure 4.12 were generated using two `Field` instances: one composed of tangent vectors acting on another composed of points.

6.6 Future Work

6.6.1 Optimization

To solve some of the limitations discussed at the beginning of this section, there is a potential solution that still avoids the need for a user-conducted optimization process: since the precomputed combinatoric functions are currently generated by Lua scripts, enabling these scripts to "inject" compiled code at run-time is a potentially good solution[35]. Such Just-In-Time (JIT) compilation would enable a user to define a new metric at runtime, and calculate the "unrolled" functions on the fly.[36] This not only adds flexibility to the platform but could increase optimization since even single basis blades could be assigned their own function table (currently the smallest type in the *Versor* program is a 3 dimensional vector). Such a loose system would be inherently more *topological*, since it would retain metric-independence and even allow for genetic algorithms that experimentally combine metric spaces to invent new n-dimensional spaces. Metrics could potentially even be adjusted on the fly. Alternative mappings would be much more available for experimention.

[35]For a nice example of live compilation, see LuaAV, a powerful runtime audio-visual processing environment by my colleagues Wesley Smith and Graham Wakefield. Indeed it is their work which suggests this kind of approach.

[36]I am indebted to my colleague Graham Wakefield for this suggestion.

6.6.2 Machine Learning and Genetic Algorithms

The Finite State Machine implementation lends itself to predicate calculus methods for predictive form-making. Given a set of states (Multivectors of varying types – say a circle, a dual line, and a flat point), an application could examine various combinations of these and find a "best fit" output state to a given problem. The appendix serves as an example to the first step in such an algorithm: given a desired output state, what are the inputs that will generate it with just one operation? Such a search space could be expanded to two or more operations, thereby generating an operation graph which could be used for genetic algorithms and morphogenesis of structures.

6.6.3 Scriptable Interface

Versor can currently be used to build cross-platform stand alone applications in C++. To be itself truly stand alone, *Versor* will require a scriptable interface for a wide range of users. Lua can be used as an extension language in this regard, offering a simple interface to the underlying code.

6.6.4 Geometric Audio

Versor links with *Gamma,* a generic signal processing library designed by my colleague Lance Putnam. This not only opens a door for filtering, spectral analysis, and granular decomposition of form, but also to geometric calculus of spatialized sound. Motors and boosts could be applied to position sound in 3D space, or to operate on the microworld of the sample buffer itself. This suggests a non-Euclidean model of spatialized audio.

6.6.5 Twistors

Building a system that can handle higher dimensional space is clearly tempting. There is, for instance, a 6D conformal model of 4D space which enables the implementation of Roger Penrose's Twistor program.

7 Conclusions

At the beginning of this thesis we outlined three issues faced by the community of GA researchers: the need for more widespread pedagogy about GA, the relative lack of integrated multimedia implementations that use GA, and the desire for more rigorous visual experimentation and exploration of GA's synthetic capacity. These issues are interconnected: all three point to a limited awareness of how the system might be employed or deployed.

To explore these unknown possibilities, *Versor* offers a performance-ready research platform for articulating complex relationships found in nature or imagined. To help develop our intuition, this project has remained focussed on abstract formal expression through *synthesis*: – the construction of surfaces, the bending and twisting of meshes, and the leveraging of various differential operators for smooth movements. By extrapolation of concepts presented in earlier works, such as linear interpolation of dual lines, we have investigated novel shape generation and animation techniques. Moving beyond screw deformations, we have a look at the effects of affine combinations of point pairs and circles, and introduced *hyper fluids* – dynamic fields that enable the animation of 3D warping phenomena. Specific examples of scientific use of the shapes and spaces described, for instance in quantum physics, artifical intelligence, and molecular modeling, are left to future work and examination of the references. Through these applications, the *analytic* power of the geometry can be explored.

Certainly the most straight-forward contribution a researcher can make to the field of Geometric Algebra is to learn it, experiment with it, extrapolate its laws, play with it, and teach it. Only with unleashed explorations will it achieve its goal of bridging disciplines. I deliberately refer to this unifying goal as belonging to the algebra itself, since transdisciplinarity seems as inherent a property of its logic as curvature is to a surface. Even just the implementation of CGA is a transdisciplinary exercise, involving higher dimensional hypercomplex mathematics, non-Euclidean geometry, Lagrangian and Eulerian physics, advanced computer programming techniques, graphics programming, and aesthetic intuitions about form and movement.

The reader is the best judge as to whether we have achieved our goal of developing an understanding of this still esoteric system. Certainly there is more work to do in teaching this algebra, but this paper has tried to combine brevity with depth in a useful way, by guiding the reader from basic Euclidean spaces through to a complete implementation of the 5 dimensional conformal model. The source code for *Versor* itself, available online[37], is heavily commented and well documented, and aims to be a reference for future engineers. Though the learning curve can feel steep, and the implementation involved, the final algebraic system can generate intricate transformations which just a few expressions, offering simple methods for the synthesis of alternative forms and the navigation of alternative spaces. Once we have entered the space described by the system, geometric intuition develops quickly and couples with the explicit guidelines of system itself. Things begin to do what one thinks they should.

One potential use of CGA's synthetic and analytic features is in the field of biological modelling. Immediately, the ease of higher-order deformations suggest a wealth of possible uses of geometric algebra, from the study of colloidal environments and biosurfaces to embryology and morphogenesis. The *orientability* of the algebra is useful for describing *chiralities* or "handedness", to which living organisms are particularly sensitive at the molecular level. Additionally, the points, point pairs, circles, and spheres that serve as the fundamental units of the conformal model, as well as the billowy results of the Möbius transformations that operate upon them, are all much better at describing natural form than the vectors are, since they can directly generate closed form solutions without need to resort to implicit surfaces or NURBS. Researchers such as David Hestenes have shown that natural laws can be compactly written in the language of GA. Bayro-Corrochano and others have shown that GA can

[37]`www.wolftype.com/versor`

also be used as a rigorous control system. Coupled with the clear shape-generating power of the algebra as demonstrated by Rosenhahn, Lasenby, Wareham, Cameron, Perwass and others, GA is a prime candidate for investigating the complex behaviors, symmetries and forms that characterize the organic world.

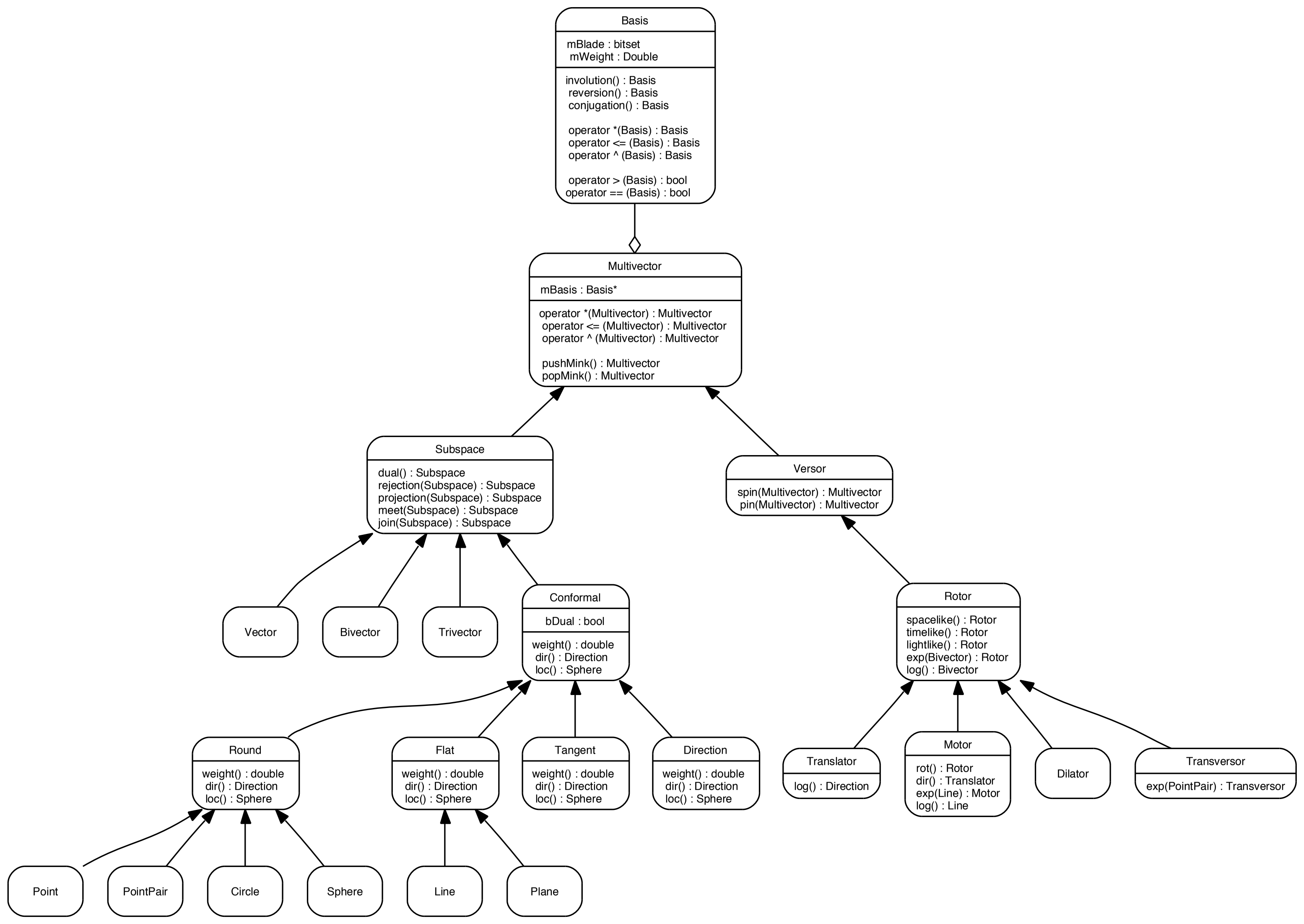


Figure 7.1: An Object-Oriented Reference Implementation of Conformal Geometric Algebra

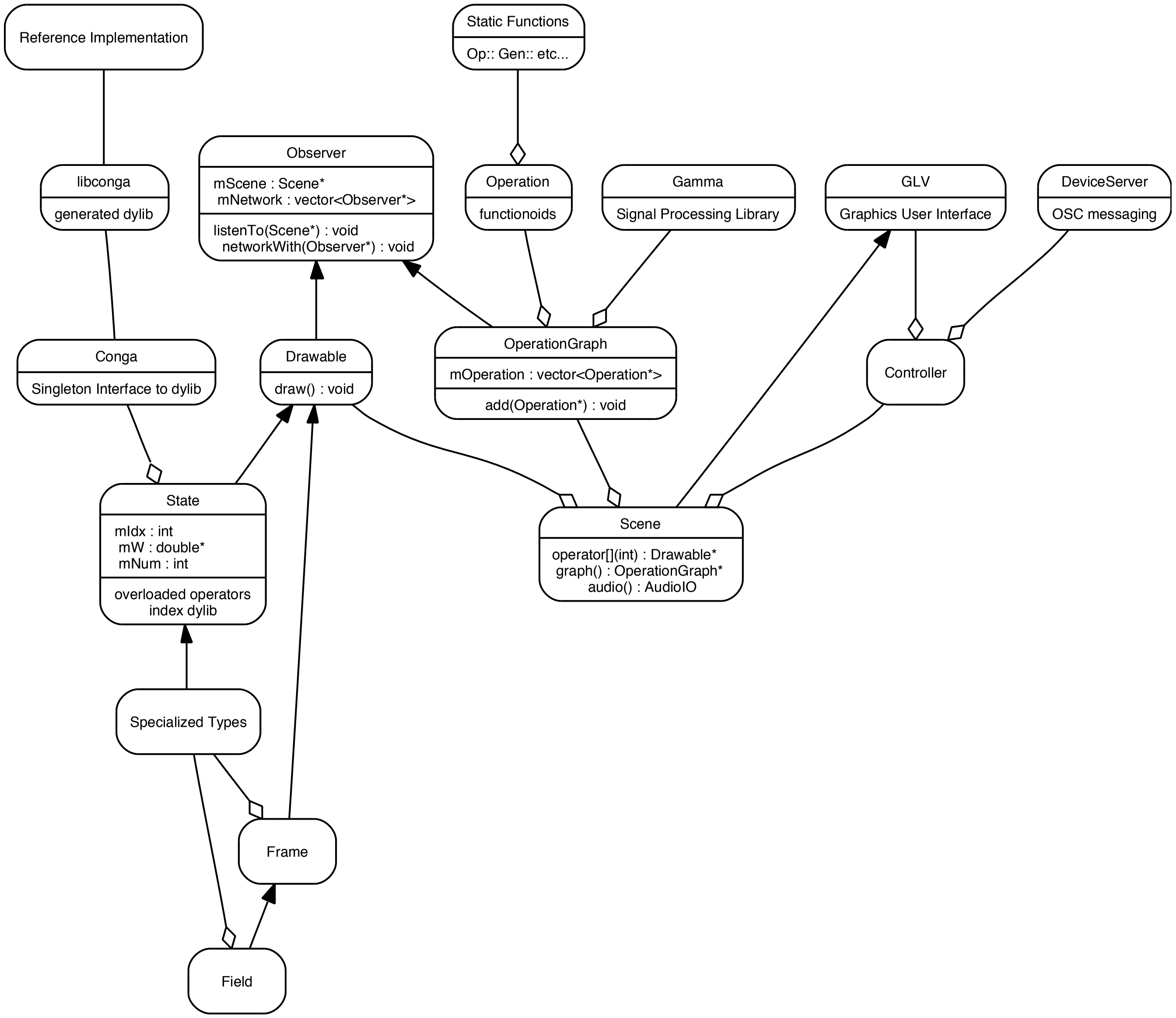
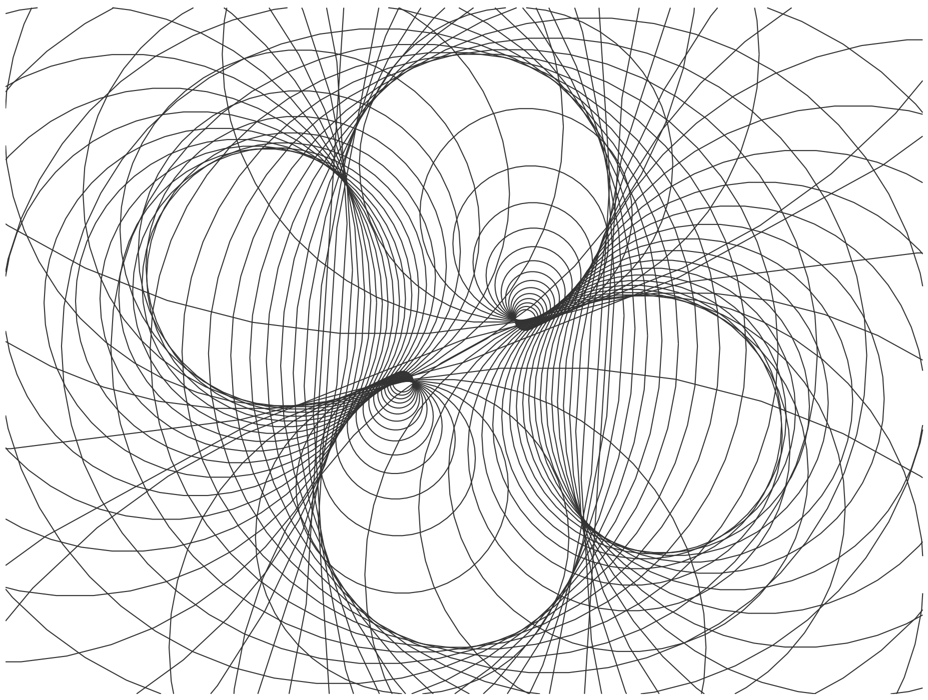


Figure 7.2: The Finite State Machine Data-Flow Implementation of Versor



References

[1] D. Hildenbrand, D. Fontijne, C. Perwass and L. Dorst, *Geometric Algebra and its Application to Computer Graphics*, 25th Annual Conference of the European Association for Computer Graphics, EUROGRAPHICS 2004.

[2] B. Rosenhahn, G. Sommer, R. Klette, *Pose Estimation of Free-form Objects*, CHRISTIAN-ALBRECHTS-UNIVERSIT¨AT KIEL, Bericht Nr. 0401 March 2004.

[3] J.M. Selig, *Lie Groups and Lie Algebras in Robotics,* South Bank University London SE1 0AA, U.K. 2003.

[4] E. Bayro-Corrochano, L. Reyes-Lozano, J. Zamora-Esquivel, *Conformal Geometric Algebra for Robotic Vision*, Journal of Mathematical Imaging and Vision 24: 55–81, 2006. Springer Science + Business Media, Inc. Netherlands.

[6] L.Dorst, D.Fontijne, S. Mann, *Geometric Algebra for Computer Science*, Morgan Kaufmann, 2007.

[7] J.Supter, *Geometric Algebra Primer*, http://www.jaapsuter.com/2003/03/12/geometric-algebra/.

[8] J.Baez, *The Octonions*, Bulletin (New Series) of the American Mathematical Society, Volume 39, Number 2, Pages 145-205, published online 2001.

[9] Hestenes, D. and Sobczyk, G. *Clifford Algebra to Geometric Calculus*. 1984.

[10] Li, H., Hestenes, D., and Rockwood, A. *Generalized Homogeneous Coordinates for Computational Geometry*. In Sommer, G., editor, *Geometric Computing with Clifford Algebra*, pp 25-58. Springer-Verlag. 2001.

[11] Rida T. Farouki, Hwan Pyo Moon, Bahram Ravani. *Minkowski Geometric Algebra of Complex Sets.* Geometriae Dedicata 85: pp 283-315, 2001. Kluwer Academic Publishers. Netherlands.

[12] B. Rosenhahn, *Pose Estimation Revisited,* Dissertation Thesis, CHRISTIAN-ALBRECHTS-UNIVERSIT¨AT KIEL, Bericht Nr. 0308 September 2003.

[14] A. Rockwood, D. Hildenbrand, "Engineering Graphics in Geometric Algebra", in *Geometric Algebra Computing.* ed. Eduardo Bayro-Corrochano, G. Scheuermann. Springer-Verlag London Limited 2010.

[15] V. Banarer, C. Perwass, G.Sommer, *Design of a Multilayered Feed-Forward Neural Network Using Hypersphere Neurons.* Institut für Informatik und Praktische Mathematik Christian-Albrechts-Universität zu Kiel, Germany. In COMPUTER ANALYSIS OF IMAGES AND PATTERNS Lecture Notes in Computer Science, Volume 2756/2003, 571-578. Springer Berlin / Heidelberg, 2003.

[16] C. Doran, A. Lasenby, *Geometric Algebra for Physicists*, Cambridge, 2003.

[17] E. Bayro-Corrochano, *Geometric Computing: For Wavelet Transforms, Robot Vision, Learning, Control and Action.* Springer Verlag, London. 2010.

[18] M. Ribeiro, C. Paiva, "Transformation and Moving Media: A Unified Approach Using Geometric Algebra". In *Metamaterials and Plasmonics: Fundamentals, Modelling, Applications.* eds S. Zouhdi, A. Sihvola, A. Vinogradov. NATO Science for Peace and Security Series B: Physics and Biophysics, 2009, Part II, 63-74.

[19] G.Sommer, B.Rosenhahn, C.Perwass, *The Twist Representation of Free Form Objects,* in Geometric Properties for Incomplete Data. eds. Klette, Reinhard and Kozera, Ryszard and Noakes, Lyle and Weickert, Joachim. Springer Netherlands. 2006, 1, pp 3-22.

[20] D. Hestenes. *New Foundations for Classical Mechanics.* D. Reidel, Dordecht/Boston, 2nd Edition, 1998.

[21] A. Naeve, L. Svennsson. *Geo-MAP Unification.* In *Geometric Computing with Clifford Algebras.* ed. G. Sommer, Spinger Verlag, Berlin Heidelberg, 2001.

[22] D. Fontijne. *Efficient Implementation of Geometric Algebra.* PhD. Thesis, University of Amsterdam, 2007.

[23] G. Sommer, ed. *Geometric Computing with Clifford Algebra.* Springer. Germany, 2001.

[24] D. Hestenes, J. Holt, *The Crystallographic Space Groups in Geometric Algebra.* Journal of Mathematical Physics, 2007.

[25] L. Dorst, *The Representation of Rigid Body Motions in the Conformal Model of Geometric Algebra.* In B. Rosenhahn et al. (eds.), *Human Motion – Understanding, Modelling, Capture, and Animation,* 507–529. Springer-Verlag. 2008.

[26] R. Wareham, J. Cameron, A. Lasenby, *Applications of Conformal Geometric Algebra in Computer Vision and Graphics.* In H. Li, P. J. Olver and G. Sommer (Eds.): 2004, Lecture Notes in Computer Science 3519, pp. 329–349. Springer-Verlag Berlin Heidelberg 2005.

[27] D. Hildenbrand, *Geometric Computing in Computer Graphics and Robotics using Conformal Geometric Algebra.* PhD Disseratation. 2006.

[28] J. Stam, *Real-Time Fluid Dynamics for Games.* Proceedings of the Game Developer Conference, March 2003.

Appendix A: List of Operators By Return Type

Operations that Construct a Point

Pss ∗ Sph	Par ⌋ Cir	Par ⌋ Lin	Cir ⌋ Sph	Cir ⌋ Pln	Dll ⌋ Cir	Dlp ⌋ Par
Vec ⌋ Par	Biv ⌋ Cir					

Operations that Construct a Sphere

Pss ∗ Pnt	Pnt ∧ Cir	Par ∧ Par	Par ∧ Dll	Par ∧ Biv	Cir ∧ Dlp	Cir ∧ Vec

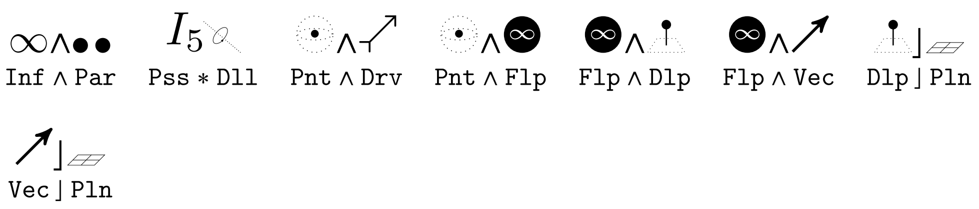
Operations that Construct a Point Pair

Pss ∗ Cir	Pnt ∧ Pnt	Pnt ⌋ Cir	Pnt ⌋ Lin	Pnt ∧ Dlp	Dll ⌋ Sph	Dlp ⌋ Cir
Vec ⌋ Cir						

Operations that Construct a Circle

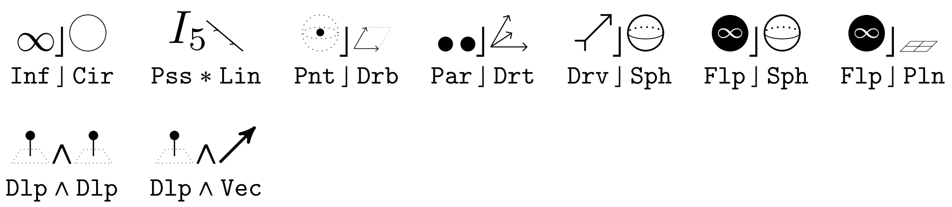
Pss ∗ Par	Pnt ∧ Par	Pnt ⌋ Sph	Pnt ⌋ Pln	Pnt ∧ Dll	Par ∧ Dlp	Par ∧ Vec
Dlp ⌋ Sph						

Operations that Construct a Line



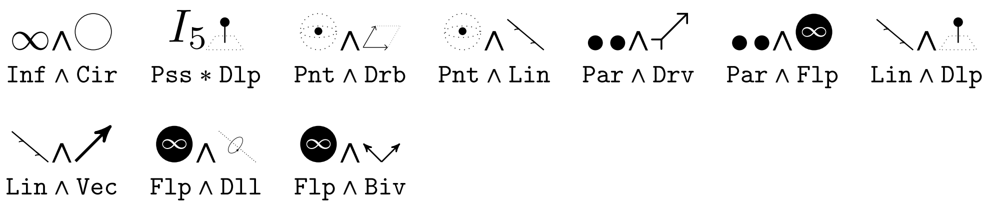
Inf ∧ Par
Pss ∗ Dll
Pnt ∧ Drv
Pnt ∧ Flp
Flp ∧ Dlp
Flp ∧ Vec
Dlp ⌋ Pln
Vec ⌋ Pln

Operations that Construct a Dual Line



Inf ⌋ Cir
Pss ∗ Lin
Pnt ⌋ Drb
Par ⌋ Drt
Drv ⌋ Sph
Flp ⌋ Sph
Flp ⌋ Pln
Dlp ∧ Dlp
Dlp ∧ Vec

Operations that Construct a Plane



Inf ∧ Cir
Pss ∗ Dlp
Pnt ∧ Drb
Pnt ∧ Lin
Par ∧ Drv
Par ∧ Flp
Lin ∧ Dlp
Lin ∧ Vec
Flp ∧ Dll
Flp ∧ Biv

Operations that Construct a Dual Plane

`Inf ⌋ Par`	`Pss ∗ Pln`	`Pnt ⌋ Drv`	`Pnt ⌋ Dll`	`Pnt ⌋ Trs`	`Par ⌋ Drb`	`Cir ⌋ Drt`
`Cir ⌋ Mot`	`Drv ⌋ Cir`	`Drb ⌋ Sph`	`Lin ⌋ Sph`	`Lin ⌋ Pln`	`Flp ⌋ Cir`	`Flp ⌋ Lin`
`Vec ⌋ Dll`						

Operations that Construct a Flat Point

`Inf ∧ Pnt`	`Mnk ∗ Trs`	`Dll ⌋ Pln`	`Dlp ⌋ Lin`	`Vec ⌋ Lin`	`Biv ⌋ Pln`

Operations that Construct a Vector

`Ori ⌋ Drv`	`Ori ⌋ Dll`	`Ori ⌋ Trs`	`Inf ⌋ Tnv`	`Inf ⌋ Trv`	`Mnk ⌋ Cir`	`Mnk ⌋ Lin`
`Pnt ⌋ Biv`	`Pnt ⌋ Rot`	`Par ⌋ Tri`	`Drv ⌋ Tnb`	`Drb ⌋ Tnt`	`Tnv ⌋ Drb`	`Tnb ⌋ Drt`
`Tnb ⌋ Mot`	`Lin ⌋ Tnt`	`Flp ⌋ Tnb`	`Dll ⌋ Tri`	`Dlp ⌋ Biv`	`Dlp ⌋ Rot`	`Biv ⌋ Tri`
`Biv ∗ Tri`						

Operations that Construct a Bivector

Ori ⌋ Drb   Inf ⌋ Tnb   Mnk ⌋ Sph   Mnk ⌋ Pln   Pnt ⌋ Tri   Drv ⌋ Tnt   Tnv ⌋ Drt

Flp ⌋ Tnt   Dlp ⌋ Tri   Vec ∧ Vec   Vec ⌋ Tri   Vec ∗ Tri

Operations that Construct a Trivector

Ori ⌋ Drt   Inf ⌋ Tnt   Mnk ⌋ Pss   Mnk ∗ Pss   Vec ∧ Biv

Operations that Construct a Direction Vector

Inf ⌋ Lin   Inf ∗ Lin   Inf ∧ Dlp   Inf ∗ Dlp   Inf ∧ Vec   Inf ∗ Vec   Pss ∗ Drb

Drb ∗ Tri   Drt ∗ Pln   Drt ∗ Dll   Drt ∗ Biv   Dlp ⌋ Drb   Vec ⌋ Drb

Operations that Construct a Direction Bivector

Inf ⌋ Pln   Inf ∗ Pln   Inf ∧ Dll   Inf ∗ Dll   Inf ∧ Biv   Inf ∗ Biv   Pss ∗ Drv

Drv ∧ Dlp   Drv ∧ Vec   Drv ∗ Tri   Drt ∗ Lin   Drt ∗ Dlp   Drt ∗ Vec

Operations that Construct a Direction Trivector						
Inf ⌋ Pss	Inf ∗ Pss	Inf ∧ Tri	Inf ∗ Tri	Drv ∧ Dll	Drv ∧ Biv	Drb ∧ Dlp
Drb ∧ Vec	Dll ∧ Dll	Dll ∧ Biv	Dlp ∧ Tri			

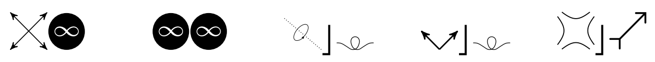
Operations that Construct a Tangent Vector at Origin					
Ori ∧ Vec	Ori ∗ Vec	Pss ∗ Tnb	Tnb ∗ Tri	Tnt ∗ Biv	Vec ⌋ Tnb

Operations that Construct a Tangent Bivector at Origin					
Ori ∧ Biv	Ori ∗ Biv	Pss ∗ Tnv	Tnv ∧ Vec	Tnv ∗ Tri	Tnt ∗ Vec

Operations that Construct a Tangent Trivector at Origin					
Ori ⌋ Pss	Ori ∗ Pss	Ori ∧ Tri	Ori ∗ Tri	Tnv ∧ Biv	Tnb ∧ Vec

Operations that Construct a Rotor			
Tnv ⌋ Mot	Vec ∗ Vec	Biv ∗ Biv	Mot ⌋ Biv

Operations that Construct a Translator



`Mnk ∗ Flp` `Flp ∗ Flp` `Dll ⌋ Mot` `Biv ⌋ Mot` `Trv ⌋ Drv`

Operations that Construct a Dilator

`Ori ∗ Inf` `Drt ∗ Tnt`

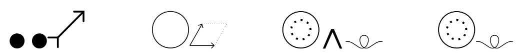
Operations that Construct a Motor

`Lin ∗ Lin` `Dll ∗ Dll` `Dll ∗ Biv` `Dll ∗ Rot` `Rot ∧ Trs` `Rot ∗ Trs`

Operations that Construct a Transversor

`Trs ⌋ Tnv` `Mot ⌋ Tnv`

Operations that Construct a Motor Dilator



`Par ∗ Drv` `Cir ∗ Drb` `Dil ∧ Mot` `Dil ∗ Mot`

Operations that Construct a Minkowski Plane

`Ori ∧ Inf` `Pss ∗ Tri`

Operations that Construct a Pseudoscalar

Ori ∧ Sph	Ori ∧ Drt	Ori ∧ Pln	Inf ∧ Sph	Inf ∧ Tnt	Mnk ∧ Cir	Mnk ∧ Tri
Mnk ∗ Tri	Pnt ∧ Sph	Pnt ∧ Drt	Pnt ∧ Tnt	Pnt ∧ Pln	Par ∧ Cir	Par ∧ Drb
Par ∧ Tnb	Par ∧ Lin	Par ∧ Tri	Cir ∧ Drv	Cir ∧ Tnv	Cir ∧ Flp	Cir ∧ Dll
Cir ∧ Biv	Sph ∧ Dlp	Sph ∧ Vec	Drv ∧ Tnb	Drb ∧ Tnv	Tnv ∧ Lin	Tnb ∧ Flp
Tnb ∧ Dll	Tnt ∧ Dlp	Lin ∧ Dll	Lin ∧ Biv	Pln ∧ Dlp	Pln ∧ Vec	Flp ∧ Tri

Operations that Construct an Origin

Pss ∗ Tnt	Tnt ∗ Tri	Vec ⌋ Tnv	Vec ⌋ Trv	Biv ⌋ Tnb

Operations that Construct an Infinity

Pss ∗ Drt	Drv ⌋ Lin	Drb ⌋ Pln	Drt ∗ Tri	Dll ⌋ Drb	Dll ⌋ Lin	Dlp ⌋ Mnk
Dlp ⌋ Drv	Dlp ⌋ Flp	Dlp ⌋ Trs	Dlp ⌋ Dil	Vec ⌋ Drv	Vec ⌋ Flp	Vec ⌋ Trs
Biv ⌋ Drb	Biv ⌋ Lin	Tri ⌋ Pln	Tri ⌋ Mot			