Computer Graphics Using
Conformal Geometric Algebra
First Year Report

Richard Wareham
<rjw57@eng.cam.ac.uk>

September 20, 2004
## Contents

1 Introduction ........................................ 4  
   1.1 Historical Note ........................................ 4  
   1.2 Introduction to Geometric Algebra ................. 5  
      1.2.1 Problems with Vector Algebra ................. 5  
      1.2.2 Clifford Algebra .................................. 5  
      1.2.3 Rotations using Clifford Algebra ............. 6  
      1.2.4 Conformal Geometric Algebra (CGA) ........... 8  
      1.2.5 Rotations ........................................ 9  
      1.2.6 Other transformations ............................ 9  
      1.2.7 Geometric objects ................................ 10  
   1.3 Existing Systems ................................... 10  
      1.3.1 CLU & CLUDraw ................................. 10  
      1.3.2 Gaigen .......................................... 11  
      1.3.3 Cambridge GA library for Maple ............... 12  

2 Implementation ........................................ 13  
   2.1 Design of libcga ................................... 13  
      2.1.1 Representation of multivectors ............... 13  
      2.1.2 Implementation details .......................... 14  
   2.2 Optimisations ....................................... 15  
      2.2.1 Grade tracking ................................. 15  

3 Algorithms using CGA .................................. 18  
   3.1 Finding Intersections .............................. 18  
      3.1.1 The method of projectors ...................... 18  
   3.2 Visualisation algorithms ........................... 20  
   3.3 Reflections ......................................... 20  

4 Non-Euclidean Geometries ............................... 22  
   4.1 Description of hyperbolic geometry ............... 22  
   4.2 2D non-Euclidean Space ............................ 23  

5 Hyperbolic Rendering Algorithms ....................... 26  
   5.1 Rendering $d$-lines .................................. 26  
      5.1.1 NURBS .......................................... 26  
      5.1.2 Calculating properties of $d$-lines .......... 27
5.2 Rendering ‘d-planes’ .................................................. 28

6 Rotor Interpolation ......................................................... 31
   6.1 Other approaches .................................................. 31
      6.1.1 Spherical linear interpolation (SLERP) ....................... 31
      6.1.2 Mullineux interpolation ..................................... 32
   6.2 The Rotor ‘logarithm’ approach ................................... 32
      6.2.1 Rotor interpolation in Euclidean geometry ................. 33
   6.3 Comparison of rotor interpolation to other techniques ......... 36

7 Future Work .............................................................. 38
   7.1 Basic Implementation ............................................... 38
   7.2 Non-Euclidean Geometries ......................................... 38
   7.3 Rotor Interpolation .................................................. 39
   7.4 Timetable for Future Work ........................................ 39
Chapter 1

Introduction

This report introduces a framework for geometric manipulation which we call the conformal model. It offers an intuitive approach to many common Computer Graphics operations both in $\mathbb{R}^3$ and higher dimensions.

Chapter 1 introduces the existing knowledge in the field of Conformal Geometric Algebra (CGA) and provides a brief discussion of the ‘toolkits’ used currently.

Chapter 2 describes the design aims and implementation details of a software package I implemented to perform numerical CGA calculations. This package was used to test and implement the algorithms subsequently described.

Chapter 3 describes common algorithms used in CGA and the method of projectors which was created to allow for the extraction of certain intersections.

Chapter 4 introduces developments in CGA towards extension into non-Euclidean geometries and includes a brief discussion of Hyperbolic space.

Chapter 5 concerns itself with the visualisation algorithms developed to draw objects within hyperbolic space and the extension to three-dimensional Hyperbolic space.

Chapter 6 discusses a novel form of rigid-body transform interpolation developed using CGA which generalises into non-Euclidian, and higher-dimension spaces.

Chapter 7 provides an overview of what was achieved within this report and discusses possible future applications.

1.1 Historical Note

Since its inception in the mid-1970s, Computer Graphics (CG) has almost universally used linear vector algebra as its mathematical framework. This is due primarily to two factors; most early practitioners of computer graphics were mathematicians familiar with it and linear algebra provided a compact, efficient way of representing points, transformations, lines, etc.

In the early-1980s CG moved out of the realm of Computer Science research and started to be used in the broader scientific community as an important research tool, both for simulation and visualisation. Computer Graphics was still tied to classical vector algebra which had started to show a number of problems when applied to the problems being investigated. Amongst these were poor generalisation to spaces other than $\mathbb{R}^3$ and great conceptual difficulty in extending problems to non-Euclidean spaces and manipulating geometric objects other than simple lines, planes and points.

As computing power becomes cheaper, the opportunity arises to investigate new frameworks for CG which, although not providing the time/space efficiency of vector
algebra, may provide a conceptually simpler system or one of greater analytical power.

1.2 Introduction to Geometric Algebra

1.2.1 Problems with Vector Algebra

As stated above, classical vector algebra has a number of problems once one moves away from three-dimensional Euclidean space. Perhaps the clearest example is the cross-product of two vectors. The product, \( \mathbf{a} \times \mathbf{b} \), is conventionally defined as a vector normal to the plane that \( \mathbf{a} \) and \( \mathbf{b} \) both lie in and has magnitude \( \mathbf{ab} \sin \theta \) where \( \theta \) is the angle between \( \mathbf{a} \) and \( \mathbf{b} \). However, the normal to the plane is only uniquely defined in 3-dimensions and has no meaning in 2- or 1-dimensional space; the cross-product does not generalise to higher-dimension spaces. The product is an important element of vector algebra and one can see that performing geometric operations in higher-spaces without it quickly becomes complex.

It was in an attempt [5] to create an algebra of vectors which did generalise to higher spaces that a German schoolteacher named Hermann Graßmann (1809–1877) created an exterior or outer product of two vectors denoted as \( \mathbf{a} \wedge \mathbf{b} \). Graßmann’s outer product can be visualised geometrically as moving one vector along the other to form a ‘directed area’. This is a new object, neither a vector or a scalar. We shall refer to it as a bivector. Similarly one may form the outer product of this bivector with another vector to form a directed volume or trivector. We say a bivector has grade 2, a trivector has grade 3 and, generally, a \( n \)-vector has grade \( n \).

The geometric objects are illustrated in Figure 1.1. A key feature is that the outer-product is anti-commutative and associative so that

\[
\mathbf{a} \wedge \mathbf{b} = -(\mathbf{b} \wedge \mathbf{a}) \quad \text{and} \quad \mathbf{a} \wedge (\mathbf{b} \wedge \mathbf{c}) = (\mathbf{a} \wedge \mathbf{b}) \wedge \mathbf{c} = \mathbf{a} \wedge \mathbf{b} \wedge \mathbf{c}
\]

1.2.2 Clifford Algebra

Most of Graßmann’s work was largely ignored by the mathematical community. It was not until William Clifford (1845–1879) investigated Graßmann’s algebra in 1878 [3] that the crucial step was made.

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

\[
\mathbf{a} \cdot \mathbf{b} + \mathbf{a} \wedge \mathbf{b} = 0
\]

A second glance at this shows an interesting feature which should be noted. In the expression above we are adding a scalar \( \mathbf{a} \cdot \mathbf{b} \) to a bivector \( \mathbf{a} \wedge \mathbf{b} \). This combination of
different objects is analogous to complex numbers where we linearly combine a real and imaginary number to form a complex number. In this case we refer to a linear combination of objects of varying grade as a multivector. Since the algebra may be extended to deal with multivectors we no longer need different notation for scalars, vectors, bivectors, etc. This project will use the convention of using lower-case letters to refer to single-grade objects and upper-case letters to refer to multivectors. The geometric product also gives us a convenient new definition of the outer and inner products for vectors

\[ A \wedge B = \frac{1}{2}(AB - BA) \]

\[ A \cdot B = \frac{1}{2}(AB + BA) \]

The power of this approach may be illustrated through its application to rotation. In 2-D this is easily performed using complex numbers. Representing the vector \([x \ y]\) as the complex number \(z = x + iy\), rotation by \(\theta\) radians can be performed by multiplication with \(e^{i\theta}\). William Hamilton (1805–1865) worked for many years to extend this approach to 3-D. He eventually created quaternions \([6, 7]\), an algebra with 4 elements \(\{1, i, j, k\}\) which, although functional, lacked an obvious geometrical interpretation and again didn’t generalise easily to higher-dimensions.

### 1.2.3 Rotations using Clifford Algebra

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

\[ B_1 = e_2e_3, \quad B_2 = e_3e_1, \quad B_3 = e_1e_2 \]

Note that these are indeed bivectors since

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

Now consider the effect of \(B_3\) on the vectors \(e_1\) and \(e_1 + e_2\):

\[ e_1B_3 = e_1e_1e_2 = e_1^2e_2 = e_2 \]
\[ (e_1 + e_2)B_3 = e_1B_3 + e_2B_3 = e_2 + e_2e_1e_2 = e_2 - e_1e_2^2 = e_2 - e_1 \]

By looking at Figure 1.2 it is clear that \(B_3\) has the effect of rotating the vectors counter-clockwise by 90 degrees. It is, in fact, a general property that the bivector \(e_i e_j\) will rotate a vector 90 degrees in the plane defined by \(e_i\) and \(e_j\). At first glance this
seems to offer little more than quaternions but at no point have we assumed that we are working in 3-dimensional space; this method also works in higher-dimension spaces.

Now we extend to general rotations. Firstly it is trivial to show that $B_3$ squares to $-1$:

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

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

$$Z = r(e_1 \cos \theta + e_2 \sin \theta) = e_1 r(\cos \theta + B_3 \sin \theta)$$

where $r$ is the distance of the point $Z$ from the origin and $\theta$ is the angle $Z$ makes to $e_1$. Also note that $e_1 B_3 = e_1^2 e_2 = e_2$. By taking the Taylor expansion of cosine and sine and re-arranging the coefficients it can be shown that

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

which is the analogous form of de Moivre’s theorem for complex numbers.

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

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

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

$$Z' = Ze^{B_3 \phi} = Z(\cos \phi + B_3 \sin \phi)$$

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

Careful consideration must be given to the case where the vector to be rotated, $x$, does not lie on the plane of rotation as in Figure 1.3. Firstly decompose the vector into a component which lies in the plane $x_\parallel$ and one normal to the plane $x_{\perp}$

$$x = x_\parallel + x_{\perp}$$
Now consider the effect of the following

\[ e^{-B\phi/2}x e^{B\phi/2} = \left( \cos \frac{\phi}{2} - B_3 \sin \frac{\phi}{2} \right) (x_\parallel + x_\perp) \left( \cos \frac{\phi}{2} + B_3 \sin \frac{\phi}{2} \right) \]

\[ = x_\parallel (\cos \phi + B_3 \sin \phi) + x_\perp \]

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

This leads to a general method of rotation in any plane; we form a bivector of the form \( R = e^{-B\phi/2} \) for a given rotation \( \phi \) in a plane specified by the bivector \( B \) and the transformation is given by

\[ x \mapsto RxR^{-1} \]

We refer to these bivectors which have a rotational effect as rotors. Figure 1.3 shows the various objects used.

The computation of \( R^{-1} \) is rather difficult analytically (and indeed can require a full \( 2^n \)-dimension matrix inversion for a space of dimension \( n \)). To combat this we define the reversion of a \( n \)-vector \( X = e_ie_j...e_k \) as

\[ \bar{X} = e_k...e_je_i \]

i.e. the literal reversion of the components. By looking at the expression for \( R \) it is clear that \( \bar{R} \equiv R^{-1} \) for rotors. Computing \( \bar{R} \) is easier since it is simply a permutation.

Note that in spaces with dimension \( n \) the maximum grade object possible is an \( n \)-grade one. We denote the \( n \)-vector \( e_1 \land ... \land e_n = I \) as the pseudoscalar and the product \( xI \) as the dual of a vector or \( x^\ast \). The dual is also defined for general multivectors.

### 1.2.4 Conformal Geometric Algebra (CGA)

In the Conformal Model [8] we extend the space by adding two additional basis vectors. We first define the signature, \( (p, q) \) of a space \( A(p, q) \) with basis vectors, \( \{e_i\} \), such that \( e_i^2 = +1 \) for \( i = 1, ..., p \) and \( e_j^2 = -1 \) for \( j = p + 1, ..., p + q \). For example \( \mathbb{R}^3 \) would be denoted as \( A(3, 0) \). We extend \( A(3, 0) \) so that it becomes mixed signature and is defined by the basis

\[ \{e_1, e_2, e_3, e, \bar{e}\} \]

where \( e \) and \( \bar{e} \) are defined so that

\[ e^2 = 1, \quad \bar{e}^2 = -1, \quad e \cdot \bar{e} = 0 \]

\[ e \cdot e_i = \bar{e} \cdot e_i = 0 \quad \forall \ i \in \{1, 2, 3\} \]

This space is denoted as \( A(4, 1) \). In general a space \( A(p, q) \) is extended to \( A(p+1, q+1) \). We now consider the vectors \( n \) and \( \bar{n} \) where

\[ n = e + \bar{e}, \quad \bar{n} = e - \bar{e} \]

It is simple to show by direct substitution that both \( n \) and \( \bar{n} \) are null vectors (i.e. \( n^2 = \bar{n}^2 = 0 \)).

A vector \( x \) in \( A(3, 0) \) can be mapped to a vector \( F(x) \) in \( A(4, 1) \) using the Hestenes mapping [9].

\[ F(x) = x^2n + 2x - \bar{n} \]
Again, by simple substitution, it is easy to show that \([F(x)]^2 = 0\) so all vectors \(x\) in \(A(3,0)\) can be mapped onto null-vectors in \(A(4,1)\). From now on lowercase letters will be used for vectors in \(A(3,0)\) and uppercase letters for vectors and multivectors in \(A(4,1)\).

Before we proceed it is worth looking carefully at \(F(x)\). Firstly, we could specify the inverse-mapping to be independent of absolute scale, i.e. \(F^{-1}(\alpha F(x)) = x \forall \alpha \neq 0\). This would be fine if we only ever combined these multivectors using wedge or geometric products, where scale has no effect on the direction of the result. However, to allow us to always add the multivectors and recover some \(x\) we impose the (arbitrary) normalisation constraint

\[
F(x) \cdot n = -1
\]

Which leads to a re-definition of \(F(x)\):

\[
F(x) = \frac{1}{2}(x^2 n + 2x - \bar{n})
\]

We can further note that the equation is dimensionally inconsistent. To restore dimensional consistency we introduce a fundamental length scale \(\lambda\) with units of length:

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

For Euclidean geometry we usually take \(\lambda\) to be unity but its importance will become clear when considering non-Euclidean geometries.

1.2.5 Rotations

A useful property of this mapping is that pure-rotation rotors retain their properties. This can be shown by considering the effect of a rotor \(R\) upon \(F(x)\)

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

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

\[
= F(Rx\tilde{R})
\]

Note that we have used the property that rotors leave \(n\) and \(\bar{n}\) invariant. This defining feature of Euclidean space points us in the direction of how to generalise this method to non-Euclidean geometries.

1.2.6 Other transformations

A similar approach used to derive the forms of a pure-rotation rotor allows us to derive a rotor \(T_a = \exp(na/2)\) which has the effect of translating the vector \(x\) along \(a\), that is to say

\[
T_a F(x)\tilde{T}_a \equiv F(x + a)
\]

It can also be shown that the rotor \(D_\alpha = \exp(\alpha e\bar{e}/2)\) has the effect of dilating \(x\) by a factor of \(e^{-\alpha}\):

\[
D_\alpha F(x)\tilde{D}_\alpha \propto F(e^{-\alpha}x)
\]

Finally inversions \((x \mapsto -x)\) may be represented as \(F(x) \mapsto eF(x)e\) [10]. This is equivalent to a reflection in \(e\). Reflections will be discussed later.
1.2.7 Geometric objects

So far all the operations discussed may be performed using Clifford algebra in $\mathbb{R}^3$ or indeed using classical linear algebra with a suitable projective geometry (such as homogeneous co-ordinates). The advantage of CGA when compared to these approaches is the unified and analytically-efficient way of performing certain geometric operations. We also have a way to deal with geometric objects, lines, spheres, planes, etc, as efficiently and easily as if they were points.

It can be shown that a blade (e.g. bivector, trivector, etc) $M$ can be found such that the solutions to

$$X \wedge M = 0$$

all lie on a circle, line, plane or sphere. The form of $M$ is the wedge-product of a set of defining points. We use the symbol $\wedge$ to denote the wedge-product in an analogous manner to the way $\sum$ is used for summation, i.e.

$$\bigwedge_{i \in \{1,2,3\}} X_i = X_1 \wedge X_2 \wedge X_3$$

The defining points for various objects are summarised in table 1.1.

Notice that planes and spheres are both grade-4 objects and circles and lines are grade-3 objects. An interesting observation is that the expression for a circle passing through the point $n$ is the same as a line and a sphere passing through $n$ is the same as a plane. This indicates that $n$ may be identified with the point-at-infinity in $\mathcal{A}(4,1)$. Recall that in our discussion of rotors a required property was that $Rn \bar{R} = n$, i.e. that rotors leave the point at infinity invariant which is one defining property of Euclidean geometry. Another is that rotors must leave the origin invariant. The condition that $R \bar{n} \bar{R} = \bar{n}$ also suggests that $\bar{n}$ can be identified with the origin. This is clear when one notes that $F(0) \propto \bar{n}$.

1.3 Existing Systems

One of the goals of this PhD was to continue to develop a software library designed to help with the implementation of CGA-based algorithms in an efficient manner. Before work started on designing the software, several existing systems were investigated. All of the following packages were designed to provide high-level access to numerical computations using GA.

1.3.1 CLU & CLUDraw

CLU & CLUDraw were written by Christian Perwass and may be obtained from his web-site (http://www.perwass.de/cbup/clu.html). Of all the systems, this is the only one designed both for CGA and the visualisation of spheres, circles, etc. directly from the CGA model.
Table 1.2: Example \TeX{} output from Gaigen

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>$e_1$</th>
<th>$e_2$</th>
<th>$e_3$</th>
<th>$e_{12}$</th>
<th>$e_{13}$</th>
<th>$e_{23}$</th>
<th>$e_{123}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+1</td>
<td>+$e_1$</td>
<td>+$e_2$</td>
<td>+$e_3$</td>
<td>+$e_{12}$</td>
<td>+$e_{13}$</td>
<td>+$e_{23}$</td>
<td>+$e_{123}$</td>
</tr>
<tr>
<td>$e_1$</td>
<td>+$e_1$</td>
<td>+1</td>
<td>+$e_{12}$</td>
<td>+$e_{13}$</td>
<td>+$e_2$</td>
<td>+$e_{23}$</td>
<td>+$e_{123}$</td>
<td>+$e_{23}$ +$e_{12}$</td>
</tr>
<tr>
<td>$e_2$</td>
<td>+$e_2$</td>
<td>-$e_{12}$</td>
<td>+1</td>
<td>-$e_1$</td>
<td>-$e_{123}$</td>
<td>+$e_3$</td>
<td>-$e_{13}$</td>
<td>+$e_{12}$</td>
</tr>
<tr>
<td>$e_3$</td>
<td>+$e_3$</td>
<td>-$e_{13}$</td>
<td>-$e_{23}$</td>
<td>+1</td>
<td>+$e_{123}$</td>
<td>-$e_1$</td>
<td>-$e_2$</td>
<td>+$e_{12}$</td>
</tr>
<tr>
<td>$e_{12}$</td>
<td>+$e_{12}$</td>
<td>-$e_2$</td>
<td>+$e_1$</td>
<td>+$e_{123}$</td>
<td>-$e_{23}$</td>
<td>-$e_1$</td>
<td>-$e_2$</td>
<td>+$e_{12}$</td>
</tr>
<tr>
<td>$e_{13}$</td>
<td>+$e_{13}$</td>
<td>-$e_3$</td>
<td>-$e_{123}$</td>
<td>+$e_1$</td>
<td>+$e_{23}$</td>
<td>-$e_{12}$</td>
<td>-$e_1$</td>
<td>-$e_2$</td>
</tr>
<tr>
<td>$e_{23}$</td>
<td>+$e_{23}$</td>
<td>+$e_{123}$</td>
<td>-$e_3$</td>
<td>+$e_2$</td>
<td>-$e_{13}$</td>
<td>+$e_{12}$</td>
<td>-$e_1$</td>
<td>-$e_2$</td>
</tr>
<tr>
<td>$e_{123}$</td>
<td>+$e_{123}$</td>
<td>+$e_{23}$</td>
<td>-$e_{13}$</td>
<td>+$e_{12}$</td>
<td>-$e_3$</td>
<td>+$e_2$</td>
<td>-$e_1$</td>
<td>-$e_2$</td>
</tr>
</tbody>
</table>

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

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

Although it provides a convenient interface, the heavy use of C++ object-orientation and operator overloading within the library results in a rather high computational overhead. The decoupling of the calculation engine and visualisation engine however provides the useful ability to remove the graphics code in a clean manner.

### 1.3.2 Gaigen

The Gaigen homepage\(^1\) describes it as

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

The user can select the signature of the space and generate C-code to implement it. The basic code is quite naïve (similar to the initial code created for this project, see later) insofar as it performs the full $O(1024)$ matrix multiplication required. It does allow the user to specify a number of special purpose optimised routines to find the geometric product of, for example, a bivector and trivector. Gaigen doesn’t possess a visualisation engine by default.

Gaigen does have the useful ability to generate product tables for the algebra (see Table 1.2) in both plain text and \TeX{} format.

\(^1\)http://carol.wins.uva.nl/~fontijne/gaigen/
1.3.3 Cambridge GA library for Maple

This library is available from the Cavendish Astrophysics web-site\(^2\). It provides Geometric Algebra capabilities for the Maple V and VI symbolic mathematics package. It provides no visualisation capabilities above those provided by Maple. This is a very useful tool for research but is aimed more at symbolic manipulation than numerical computation.

\(^2\)http://www.mrao.cam.ac.uk/~clifford/software/
Chapter 2

Implementation

2.1 Design of libcga

There were two broad design aims when designing the libcga software library to perform numerical GA computations.

- Create a fast, efficient, user-friendly CGA library.
- Investigate geometric operations within CGA using the library.

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

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

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

2.1.1 Representation of multivectors

Any multivector \( M \) in \( \mathcal{A}(p,q) \) can be formed from a linear combination of all possible basis vector products up to grade \( p + q \). For \( \mathcal{A}(4,1) \), the highest grade object is \( e_1e_2e_3e\bar{e} \) so

\[
M = a_1 + a_2e_1 + a_3e_2 + ... + a_7e_1e_2 + ... + a_{32}e_1e_2e_3e\bar{e}
\]

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

The various multivector products may be found by expanding out the terms of the product and simplifying using a product table similar to Table 1.2. Alternatively, one
\[ A^G = \begin{bmatrix}
+A_0 & +A_1 & +A_2 & +A_3 & -A_{12} & -A_{13} & -A_{23} & -A_{123} \\
+A_1 & +A_0 & +A_{12} & +A_{13} & -A_2 & -A_3 & -A_{123} & -A_{23} \\
+A_2 & -A_{12} & +A_0 & +A_{23} & +A_1 & +A_{123} & -A_3 & +A_{13} \\
+A_3 & -A_{13} & -A_{23} & +A_0 & -A_{123} & +A_1 & +A_2 & -A_{12} \\
+A_{12} & -A_2 & +A_1 & +A_{123} & +A_0 & +A_{23} & -A_{13} & +A_3 \\
+A_{13} & -A_3 & -A_{123} & +A_1 & -A_{23} & +A_0 & +A_{12} & -A_2 \\
+A_{23} & +A_{123} & -A_3 & +A_2 & +A_{13} & -A_{12} & +A_0 & +A_1 \\
+A_{123} & +A_{23} & -A_{13} & +A_{12} & +A_3 & -A_2 & +A_1 & +A_0 
\end{bmatrix} \]

Figure 2.1: Example product matrix for the geometric product in \( A(3,0) \). \( A_{ij...k} \) is the element of \( A \) proportional to \( e_i e_j ... e_k \).

may view the multivectors \( A, B, C \) as the vector representations \( A, B, C \) and calculate \( C = AB \) using

\[
C = A^G B
\]

where \( A^G \) is a 32 \( \times \) 32 matrix whose elements depend on the operator used (in this case the geometric product) and the elements of \( A \). A little thought makes it clear that all linear operators can be expressed in this form and so all the operators we have defined can thus be expressed. An example matrix for the geometric product in \( A(3,0) \) generated by Gaigen is shown in Figure 2.1.

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

### 2.1.2 Implementation details

Product tables were generated with a Perl\(^1\) script. \( n \)-vector components were represented by a string of digits. For example, the trivector \( e_1 e_4 e_5 \) was represented as ‘145’. The resulting geometric product was calculated using the algorithm represented in figure 2.2. The algorithm can be outlined as followed:

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

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

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

---

\(^1\)Practical Extraction and Report Language—see http://www.perl.com/
Figure 2.2: Example of finding that $e_{145}e_{452} = e_{12}$ with $e_2^2 = -1$, $e_4^2 = 1$.

Table 2.1: Multiplication count for finding the geometric product of an $r$-vector and $s$-vector.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r$</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>10</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>5</td>
<td>25</td>
<td>50</td>
<td>50</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>10</td>
<td>50</td>
<td>100</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>50</td>
<td>100</td>
<td>100</td>
<td>50</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10</td>
<td>50</td>
<td>50</td>
<td>25</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

2.2 Optimisations

2.2.1 Grade tracking

Table 2.1 show the number of floating-point multiplications to compute the geometric product of a pure $r$-vector pure $s$-vector. As you can see even for a worst case bivector-bivector product only 100 multiplications are required compared to 1024 for the general product. This suggests that significant speedups can be obtained if only those grades present in the multivector are considered.

The script used to generate the product tables above could also be used to generate specific $n$-vector, $m$-vector product tables. These specific tables often used far fewer flops than required for the general operator. This effectively exploited the sparseness the product matrix for single-grade products.

When looking for an optimisation strategy I wanted the following properties

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

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

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

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

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

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

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

hence it can be said that

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

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

In order to implement this it was necessary to record in the multivector which grades were present. The most time and space efficient method to do this in C is via a bit-mask. This technique keeps a 32-bit unsigned integer called the grade mask. If bit $n$ in the mask is set then grade $n$ is present. This does limit the maximum grade to 32-bits but this can be extended to 64-grade by using a 64-bit integer on certain systems (e.g. the IA-64 next generation Intel processor) or even higher order grades by utilising multi-word masks.

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

The product function itself was modified to check for the presence of each grade-pair in the two input multivectors and only call the appropriate set of single-grade routines. It was the job of each single-grade product function to set the appropriate grade mask bits in the output multivector. This approach is shown diagrammatically in Figure 2.3
Figure 2.3: The method of grade tracking represented graphically. The shaded numbers represent the grades present in each multivector.
Chapter 3

Algorithms using CGA

3.1 Finding Intersections

As previously stated we can find representations for geometric objects in terms of the wedge-product of defining points. We can also define the meet operator to give an object which is the intersection of two objects, that is its wedge-product null-space is the intersection of the null-spaces of the two objects.

The general form of the meet can be found by noting that for \( r \)-grade and \( s \)-grade blades \( M_r \) and \( M_s \), a point, \( X \), on the intersection must satisfy

\[
X \wedge M_r = X \wedge M_s = 0
\]

which can then be shown to be equivalent to

\[
X \wedge \left[ \langle (M_r, M_s)_{2(l-r-s)} \rangle^* \right] = 0
\]

where \([\ ]^*\) is the dual operator, \( l \) is the dimension of the space (in the case of \( A(4,1) \), \( l = 5 \)) and \( \langle X \rangle_i \) denotes the extraction of the \( i \)-grade component from \( X \). If we define the meet operator

\[
M_r \lor M_s = \left[ \langle (M_r, M_s)_{2(l-r-s)} \rangle \right]^*
\]

Then we can interpret the meet of two objects as their intersection.

The key feature of this approach is that we have placed few constraints on the form of \( M_r \) and \( M_s \) and thus we can intersect objects in a fairly general manner instead of using object-specific algorithms. For example the classical algorithm for general sphere-circle intersection is very different to the algorithm for general sphere-line intersection (the former is far harder for example). With this approach the analytical steps are identical (as are, from the software point of view, the underlying computations).

Another useful feature of this approach is that it is not tied to the origin. In classical geometric methods one generally has to transform the problem to the origin where it may be solved analytically (often by solving directly the non-linear simultaneous equations for the intersection points leading to a large number of ‘special cases’).

Once the intersection of two objects is found, the visualisation engine uses some basic algorithms to extract the various parameters of the objects. Outlines of these algorithms will be given later but a full description can be found in [10].

3.1.1 The method of projectors

Some intersections, namely line-sphere and line-circle give, in general, two points of intersection. In these cases we find that the meet generates a new class of object, a bivector
representing a pair of points; the points $A$ and $B$ are simply represented by $A \wedge B$. It is simple to show that the null-space of this has the required form since

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

is only true in general if $B = X$ and, by symmetry, if $A = X$.

Extracting $A$ and $B$ from $A \wedge B$ requires a little care and may be performed using the method of projectors.

Let our bivector be $T = A \wedge B$. Firstly consider the effect of taking the inner product of $T$ with $n$

$$T \cdot n = -n \cdot T = -(n \cdot A)B + (n \cdot B)A = B - A$$

which is true as the identity $n \cdot F(x) = -1$ can be shown by direct substitution. Now form

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

where $\beta = T^2$ and hence $F^2 = 1$ (we shall consider the special case $\beta = 0$ later). We now define two projector operators in terms of $F$

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

$$\tilde{P} = \frac{1}{2}(1 - F)$$

As suggested by the operator naming, $\tilde{P}$ is the reverse of $P$ (the reverse of a bivector $F$ is simply $-F$). These operators also have the following interesting property

$$P^2 = \frac{1}{4}(1 + F)(1 + F)$$

$$= \frac{1}{4}(1 + 2F + F^2)$$

$$= \frac{1}{4}(2 + 2F)$$

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

and similarly $\tilde{P}^2 = \tilde{P}$. It is also trivial to show that $P \tilde{P} = 0$ and that

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

$$PB = B, \quad \tilde{P}A = A$$

We can now extract $A$ and $B$ from $T$

$$P[T \cdot n] = P(B - A) = PB - PA = B$$

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

Remember that this approach was only valid if $\beta^2 \neq 0$. Let’s consider the form of $\beta_2$ a little more

$$\begin{align*}
\beta &= (A \wedge B)(A \wedge B) \\
&= \frac{1}{4}(AB - BA)(AB - BA) \\
&= \frac{1}{4}(ABAB - ABBA - BAAB + BABA) \\
&= 0 \quad \text{iff } ABAB + BABA = ABBA - BAAB
\end{align*}$$
The condition for $\beta = 0$ can be simplified further

\[
ABAB + BABA = ABBA - BAAB \\
= AB^2A - BA^2B \\
= 0 - 0 = 0 \\
\Rightarrow ABAB = -BABA
\]

This condition is only satisfied in general if $A = B$ and hence if there is only one point of intersection, i.e. the line is tangential to the sphere. A similar type of analysis can be used to show that there is no intersection if $\beta^2 < 0$.

We thus have a general method for factorising $A \wedge B$.

### 3.2 Visualisation algorithms

Many of the algorithms developed using CGA have been used to aid visualisation of objects using the associated libcgadraw library.

For example, an algorithm was used to extract the centre, radius and plane of a multivector representing a circle for subsequent display.

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

\[
P = C \wedge n
\]

To find the radius and centre of the circle we use the alternate form of the circle \[10\]

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

where $C^* \equiv CI$ here (the dual of $C$ with respect to the pseudoscalar for the plane), $B$ is the centre of the circle and $\rho$ is the radius. The radius can be found simply using

\[
(C^*)^2 = \left(B - \frac{1}{2}\rho^2 n\right)\left(B - \frac{1}{2}\rho^2 n\right) \\
= -\rho^2 B \cdot n = \rho^2
\]

using $B^2 = n^2 = 0$ and $B \cdot n = -1$. Similarly we can find $B$ via

\[
B = C^* \left(1 + \frac{1}{2}C^* n\right)
\]

### 3.3 Reflections

An important algorithm which may have more wide-scale applications is reflection of one object in another. This is achieved by ‘sandwiching’ the object to reflect between the object it is reflected in.

For example, to ‘reflect’ a point in a plane, we sandwich it between point $P$ in a sphere $\Sigma$ to give $P'$ we use

\[
P' = \Sigma P \Sigma
\]

These reflections can prove useful since a large class of isometries can be built up with reflections. For example, rotation in a plane can be shown to be equivalent to reflection in two lines. Similarly we can work on problems which require access to the point at infinity by reflecting the problem in the unit circle, providing a dual-space where the point at infinity is mapped to the origin.
Note that we can now find an alternate form for the centre of a circle, reflecting the point at infinity in the circle gives its centre:

\[ c = C n C \]

This method also works for spheres and, in fact, is general to all \( n \)-spheres.
Chapter 4

Non-Euclidean Geometries

4.1 Description of hyperbolic geometry

Euclidean geometry, the geometry of the plane, is defined by Euclid’s Postulates:

1. A straight line segment can be drawn joining any two points.

2. Any straight line segment can be extended indefinitely in a straight line.

3. Given any straight line segment, a circle can be drawn having the segment as radius and one endpoint as center.

4. All right angles are congruent.

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

This last postulate is equivalent to what is known as the parallel postulate which loosely states that two parallel lines will never meet, even at infinity. Hyperbolic space is one of the simplest geometries that satisfy all but this last postulate.

Ignoring the last postulate may, initially, seem a purely intellectual exercise although the last postulate has never been proved to exist in real-space by experiment. Who is to say that parallel beams of light emitted from Earth won’t eventually converge somewhere out by Andromeda? Indeed, the ‘bending’ effect upon light by large gravitational fields almost guarantee the convergence of some light beams.

Hyperbolic geometry is usually represented in two dimensions on the Poincaré disc. Here the boundary circle of the disc represents the points at infinity. Everywhere outside the disc is inaccessible to the hyperbolic geometry. Straight lines, called $d$-lines by Brannan et al. [2], are represented by circular arcs which erupt normal to the boundary circle. Perhaps the most famous example of this geometry is given in Escher’s Circle Limit series of wood prints (see figure 4.1). In these, infinite tessalations of hyperbolic space are represented mapped to the Poincaré disc. They clearly show the circular nature of $d$-lines.

We seek to find a method of representing this geometry within our conformal framework.
4.2 2D non-Euclidean Space

All rigid-body transformation (i.e. rotation and translation) rotors in the Euclidean approach above leave $n$ invariant, i.e. $RnR = n$ for all rotors $R$. They also leave $\bar{n}$ invariant. We have already identified $n$ and $\bar{n}$ with the points at infinity and the origin respectively. The question arises: ‘What if we restrict rotors to keep other vectors invariant?’ Since a geometry is defined by its congruence transformations, changing those transformations will therefore reflect a different geometry.

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

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

and thus form the rotors $R_{k\ell} = \exp\left(\frac{\alpha}{2}E_k \wedge E_\ell\right)$ with $k, \ell \in \{1, 2, 3, 4\}$. Applying them to $e$ via $R_{k\ell}e\bar{R}_{k\ell} \equiv R_{k\ell}eR_{\ell k}$ we find the bivector generators of the rotors which preserve $e$ are $\bar{e}e_1$, $\bar{e}e_2$ and $e_1e_2$. The latter just correspond to rotations in the $e_1e_2$ plane and hence, the former two must be the generators of translations.

Hence we can say that a rotor which translates the origin to the vector $x$ must be given by

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

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

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

The choice of the origin is only restricted in that it must differ from the point at infinity and must not contain components of $e_1$ or $e_2$ (to retain isotropy). Either $n$ or $\bar{n}$ is a suitable choice but we choose a multiple of $\bar{n}$ to retain compatibility with the Euclidean case.

Again, as with the Euclidean case, we wish to impose a normalization condition on
the null-vectors, \( X = F_c(x) \), such that
\[
X \cdot e = -1 \quad (4.4)
\]
where we use \( F_c(x) \) to represent the mapping defined by the geometry generated by the rotors which preserve \( e \). If this is to hold then the origin must in fact be \(-\hat{n}\).

We can now find the representation of the general point \( x \) as the translation along \( x \) of the origin. Writing \( c = \cosh \left( \frac{f(r)}{2} \right) \) and \( s = \sinh \left( \frac{f(r)}{2} \right) \)
\[
F_c(x) = T_x(-\hat{n}) \tilde{T}_x \\
= [c + e\tilde{r}s](-\hat{n})[c - e\tilde{r}s] \\
= -c^2\hat{n} + 2sc\tilde{r} + s^2
d(4.5)
\]
then letting \( C = \cosh(f(r)) \) and \( S = \sinh(f(r)) \) giving \( c^2 = (C + 1)/2, s^2 = (C - 1)/2, sc = S/2 \) and hence
\[
F_c(x) = \frac{1}{2}n(C - 1) - \frac{1}{2}\hat{n}(C + 1) + S\tilde{r} \\
= \frac{1}{2} [(C - 1)n + 2S\tilde{r} - (C + 1)\hat{n}] \\
(4.8)
\]
As required \((F_c(x))^2 = 0\) and \( F_c(x) \cdot e = -1\).

It remains to choose a sensible form for \( f(r) \). We seek to choose \( f(r) \) such that the representation \( F_c(x) \) is similar to our Euclidean representation \( F(x) \), since this will allow us to use many of the same techniques we developed for the Euclidean case. We can re-write our Euclidean representation in terms of \( r \) and \( \hat{r} \) as
\[
F(x) = \frac{1}{2\lambda^2}(r^2n + 2\lambda r\hat{r} - \lambda^2\hat{n}) \\
(4.10)
\]
If we wish that \( F_c(x) \) be similar to \( F(x) \) then we have the conditions
\[
\frac{S}{C + 1} = \frac{\sinh(f(r))}{\cosh(f(r)) + 1} = \frac{r}{\lambda} \\
(4.11)
\]
and
\[
\frac{C - 1}{S} = \frac{\cosh(f(r)) - 1}{\sinh(f(r))} = \frac{r}{\lambda} \\
(4.12)
\]
so the mapping function becomes
\[
F_c(x) = \frac{C + 1}{2\lambda^2} [x^2n + 2\lambda x - \lambda^2\hat{n}] \\
= \cosh(f(r)) + 1 [x^2n + 2\lambda x - \lambda^2\hat{n}] \\
(4.13)
\]
which has a degree of similarity to the expression for \( F(x) \). Further, assuming \( r \) and \( \lambda \) are positive, we can see from equation 4.11 that \( r < \lambda \) since \( \sinh(A) < 1 + \cosh(A) \) for all \( A \).

Given equations 4.11 and 4.12, we can eliminate \( \sinh(f(r)) \) to give
\[
\frac{\cosh(f(r)) - 1}{\cosh(f(r)) + 1} = \frac{r^2}{\lambda^2} \\
(4.15)
\]
and hence \( \cosh(f(r)) = (\lambda^2 + r^2)/(\lambda^2 - r^2) \). Substituting into either 4.11 or 4.12 gives
\[
f(r) = \sinh^{-1} \left( \frac{2\lambda r}{\lambda^2 - r^2} \right) \\
(4.16)
\]
and hence we can form the following expressions for \( \sinh(f(r)) \) and \( \cosh(f(r)) \)
\[
\sinh(f(r)) = \frac{2\lambda r}{\lambda^2 - r^2} \quad \text{and} \quad \cosh(f(r)) = \frac{2\lambda^2}{\lambda^2 - r^2} - 1 \\
(4.17)
\]
Inserting these into equation 4.10 gives the final form of the non-Euclidean mapping function

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

(4.18)

We can also show that the form of the translation rotor given in 4.3 can be written as

\[ T_x = \frac{1}{\sqrt{\lambda^2 - x^2}} (\lambda + \bar{e}x) \]  

(4.19)

By finding the metric as shown in Lasenby et al. (2002) it can be shown that this gives rise to the Poincaré disc representation of non-Euclidean geometry (see Brannan et al. [2] for a description of the Poincaré disc).

Some discussion of the relevance of \( \lambda \) is worthwhile here. Notice that in order for the translator to remain real-valued, \( x^2 \leq \lambda^2 \). We can never therefore translate the origin outside of a circle radius \( \lambda \) centred upon it. The value of \( \lambda \) effectively defines a region of inaccessible space from the origin, effectively a boundary to the geometry.

This circle corresponds directly to the unit-circle boundary in the Poincaré disc representation if \( \lambda = 1 \) and simple dilations of the Poincaré representation if \( \lambda \neq 1 \). To maintain compatibility with the Poincaré representation, we usually set \( \lambda \) to be unity.
Chapter 5

Hyperbolic Rendering Algorithms

5.1 Rendering $d$-lines

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

5.1.1 NURBS

The $d$-lines on the Poincaré disc are circular arcs (and straight lines for the special cases of lines through the origin). OpenGL, the graphics library used for the implementation, has native support for a class of curves called NURBS (Non-uniform Rational B-Splines) [13].

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

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

The basis functions are defined recursively:

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

with

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

and $t_i$ being the elements of the knot vector

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

The relation between the number of knots, $m + 1$, the degree $k$ of the functions $N_{i,k}$ and the number of control points, $n + 1$ is given by $m = n + k + 1$ [12, 14].

Clearly a large family of curves can be expressed with suitable choices of knot vectors, weights and control points leading to great flexibility. All NURBS curves share some common properties however which make them useful in Computer Graphics. A NURBS curve always stays inside the convex hull of its control points [14] and thus is is very easy to compute whether the curve will be displayed at all. Further they are tangential to the piece-wise linear interpolation of control points and the end-points, see figure 5.1.
Figure 5.1: A set of control points and a typical example of an associated NURBS curve. Note that the endpoints of the curve are tangential to $P_0P_1$ and $P_5P_6$ and that the curve is within the convex hull of the points (shaded).

Figure 5.2: NURBS-based rendering of $d$-lines. Here $O$ is the origin and $A$ and $B$ are the boundary points of the line $L$.

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

A large number of different curves can be created with different control point numbers, positions and weights. Fortunately there are a number of standard techniques to generate common curves. One such method of drawing arcs is useful to us. We use three control points; one at the start of the circular arc, one at the origin of the boundary circle and one at its end, see figure 5.2. The end-point weights are unity whereas the weight of the control point at the origin is $\cos \gamma$ where $\gamma$ is the angle $OA$ makes with $AB$.

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

This allows us to construct a NURBS representation of any $d$-line on the Poincaré disc (including diameters) and efficiently draw them using OpenGL.

5.1.2 Calculating properties of $d$-lines

The last step in drawing out $d$-lines is now finding where they intersect the boundary disc, their boundary points. Once we have these points, the drawing can be performed

---

1See http://www.ddt.pwp.blueyonder.co.uk/evgeny/Intro/NURBS.htm for a demonstration of this.
via our NURBS-based method outlined above.

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

Figure 5.3 shows the rendering of \(d\)-lines in action. Here we have three \(d\)-lines created by rotating and translating the diameter to form a hyperbolic triangle (central dark-shaded region). Each of these lines were reflected in the other two to form a set of three reflected triangles (outer light-shaded regions). This operation could be repeated to tile the space.

5.2 Rendering ‘\(d\)-planes’

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

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

The method of defining planes in hyperbolic geometry is identical to our definition in Euclidean geometry; given four points on the \(d\)-plane, \(\{x_1, ..., x_4\}\), the plane \(\Phi\) is defined
Figure 5.4: \(d\)-planes are caps of the corresponding Euclidean sphere.

as

\[
\Phi = \bigwedge_{i=1}^{4} F_e(x_i) \quad (5.1)
\]

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

\[
F_e(p) \wedge \Phi = 0
\]

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

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

where \(F(x)\) is the mapping function for Euclidean geometry. The factor \((2\lambda^2)/(\lambda^2 - x^2)\) is always scalar for any vector \(x\) and we shall represent it as the function \(s(x)\). Hence we can re-write equation 5.1 as

\[
\Phi = \bigwedge_{i=1}^{4} s(x_i)F(x_i) \quad (5.2)
\]

\[
= \left[ \prod_{i=1}^{4} s(x_i) \right] \bigwedge_{i=1}^{4} F(x_i) \quad (5.3)
\]

\[
= S(x_1, x_2, x_3, x_4) \bigwedge_{i=1}^{4} F(x_i) \quad (5.4)
\]

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

It has thus been found, without doing \textit{any} explicit calculations with the metric, that \(d\)-planes are represented in the Poincaré sphere by portions of spheres. This neatly shows the analytical simplicity that this approach provides. Figure 5.4 shows the relation between \(d\)-planes and the corresponding Euclidean sphere.
Figure 5.5: \(d\)-plane spherical cap is intersection of associated sphere and half-space to right of plane \(P\).

We can find the meet between the associated Euclidean sphere and the boundary sphere to give the circle of intersection. Inspecting figure 5.4 we see that this circle is the edge of the spherical cap corresponding to the \(d\)-plane.

The spherical cap forming the \(d\)-plane can be thought of as the intersection of the half-space containing the origin and bounded by the plane of intersection (see figure 5.5). The circle of intersection is important since we wish to extract this plane of intersection efficiently. This is trivial if we note that the circle is equivalent to the wedge-product of three points on the circumference we can form the plane of the circle by simply wedging the circle with \(n\). In summary, the plane of intersection, \(P\) can be found from the \(d\)-plane \(\Phi\) in the following manner:

\[
P = k(\Phi \lor B) \land n
\]

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

Bajaj et al. [1] provide a method of finding a suitable set of control points and NURBS parameter space clipping curve to draw spherical caps from sphere/half-space intersections.

Their approach gives a set of control points and weights that together draw a little more than one hemisphere. Circular clipping paths in the parameter space are then used to form spherical caps.

This method was used to draw the spherical caps in the implementation.
Chapter 6

Rotor Interpolation

The requirement to interpolate smoothly between different rigid-body transformations if a common problem in diverse fields of Engineering and Mathematics. Path planning for robot arm actuators and animation in Computer Graphics are applications where such interpolation is often required. Consequently, there are a number of classical (and not-so classical) techniques which have been developed to interpolate between rigid body frames.

The class of transformations that preserve length and area are often termed isometries of the geometry. For our purposes, we shall use the term ‘isometry’ and ‘rigid-body transformation’ interchangeably.

6.1 Other approaches

6.1.1 Spherical linear interpolation (SLERP)

SLERP [15] is a method for interpolating pure rotations using quaternions. Quaternions [7] are an extension of complex numbers to four dimensions. Instead of one class of imaginary numbers, multiples of $i$, there are three. These are multiples of the three orthogonal numbers $i$, $j$, and $k$. These are defined by the relation

$$i^2 = j^2 = k^2 = ijk^2 = -1$$

and form a group. The algebra on this group is termed quaternionic algebra

Rotations are specified using the quaternion

$$q = \cos \frac{\phi}{2} + (r \cdot u) \sin \frac{\phi}{2}$$

where $r$ is a unit vector pointing along the rotation axis, $u = (i, j, k)$ and $\phi$ is the rotation angle. Note the similarity to the pure-rotation rotor

$$R = \cos \frac{\phi}{2} + r^* \sin \frac{\phi}{2}$$

where $r^*$ is the bivector representing the plane normal to $r$ if we are working in three dimensions.

Rotations are performed by firstly representing a point to be rotated about the origin by a quaternion of the form $p = x \cdot u$. The rotated point $x'$ can be reconstructed from its associated quaternion, $p' = x' \cdot u$, which in turn is given by

$$p' = qpq^{-1}$$

31
Again note the similarity with the application of the pure-rotation rotor.

It is trivial to show that \( q \) is a unit quaternion and hence the family of all \( q \) live on the surface of a hypersphere. Any valid rotation is thus uniquely defined by a point on this hypersphere and pure-rotations may readily be interpolated by moving the corresponding quaternions along its surface.

It is this interpolation on the surface of a hypersphere which gives SLERP its name. The \( \text{slerp}(A, B, \lambda) \) function performs a spherical interpolation of two quaternions \( A \) and \( B \) by an amount \( \lambda \in [0,1] \) and is defined as

\[
\text{slerp}(A, B, \lambda) = A \frac{\sin [(1 - \lambda)\psi]}{\sin \psi} + B \frac{\sin \psi \lambda}{\sinh \psi}
\]

where \( \psi \) is the angle the hypersphere radii passing through \( A \) and \( B \) make at the origin.

One extension to full interpolation of isometries requires the splitting into pure-rotation and pure-translation components and interpolating separately. The interpolation of the pure-translation can simply be achieved by direct linear interpolation of the position vectors, although more sophisticated approaches such as spline curves, Bézier curves, etc exist.

Almost all methods currently used in real applications treat rotation interpolation and translation interpolation separately. Many algorithms exist for optimising paths in terms of smoothness or maximal curvature but few, if any, explicitly allow for simultaneous optimisation of rotation and translation interpolation.

### 6.1.2 Mullineux interpolation

Mullineux introduced [11] a method of isometry interpolation using Clifford algebra whereby a position vector \( p \) was mapped to the representation of a point \( P \) by

\[
P = e_0 + p
\]

where \( e_0 \) is an additional basis vector where \( e_0^2 = e^{-1} \). In the final application \( \epsilon \) is allowed to tend towards zero. Mullineux then shows how to construct a multi-vector \( S \) such that \( \bar{S}PS \) represents a given rigid-body transform and further shows that all linear combinations of valid \( S \) also represent real rigid-body transforms. Hence direct component-wise interpolation of \( S \) will give valid interpolations of the rigid-body transforms.

Although he shows it to be mathematically valid, Mullineux makes no attempt to provide a justification for the form of \( S \), neither does he attempt to explain its geometric significance.

### 6.2 The Rotor ‘logarithm’ approach

Referring back to our discussion of rotors in the introduction to this report, we see that all of them have a common form. They are all exponentiated bivectors. Rotations are generated by bivectors with no component parallel to \( n \) and translations by a bivector with no components perpendicular to \( n \). We may thus postulate that all rotors can be expressed as

\[
R = \exp(B)
\]

where \( B \) is the sum of two bivectors, one formed from two vectors with no components of \( n \), call them \( a \) and \( b \), and the other with only components of \( n \). We shall therefore proceed assuming that all rotors can be written as the exponentiation of a bivector of the form \( B = ab + cn \) where \( a \), \( b \) and \( c \) are independent of \( n \). It is clear that the set of all \( B \) is some linear sub-space of all the bivectors.
We now suppose that we may interpolate rotors by defining some function $\ell(R)$ which acts upon rotors to give the generating bivector element and performing direct interpolation of these generators. It is a defining property of $\ell(R)$ that

$$R \equiv \exp(\ell(R)) \tag{6.1}$$

and so $\ell(R)$ may be considered as to act as a logarithm-like function in this context. It is worth noting that $\ell(R)$ does not possess all the properties usually associated with logarithms. Notably that, since $\exp(A) \exp(B) \neq \exp(B) \exp(A)$ generally in non-commuting algebras, $\ell(\exp(A) \exp(B)) \neq A + B$ in all cases.

To avoid the the risk of assigning more properties to $\ell(R)$ than are due it, we shall resist the temptation to denote the function $\log(R)$.

### 6.2.1 Rotor interpolation in Euclidean geometry

To interpolate Euclidean isometries we seek a method of evaluating $\ell(R)$ when applied to Euclidean rotors. The method presented below can readily be adapted to non-Euclidean rotors but the Euclidean case provides a simple exposition of the method. Having already justified the claim that all Euclidean rotors are generated by bivectors of the form $B = ab + cn$ we proceed to seek a form for $\ell(ab + cn)$.

**Theorem 1** If $B$ is of the form $B = ab + cn$ where $a, b, c$ are vectors orthogonal to $n$ and $a \perp b$ then, for any $k \in \mathbb{Z}^+$,

$$B^k = (ab)^k + \alpha_k^{(1)} abcn + \alpha_k^{(2)} abcnab + \alpha_k^{(3)} cnab + \alpha_k^{(4)} cn$$

with the following recurrence relations for $\alpha_k^{(i)}, k > 0$

$$\begin{align*}
\alpha_k^{(1)} &= \alpha_{k-1}^{(2)} (ab)^2 \\
\alpha_k^{(2)} &= \alpha_{k-1}^{(1)} \\
\alpha_k^{(3)} &= \alpha_{k-1}^{(4)} \\
\alpha_k^{(4)} &= \alpha_{k-1}^{(3)} (ab)^2 + (ab)^{k-1}
\end{align*}$$

with $\alpha_0^{(1)} = \alpha_0^{(2)} = \alpha_0^{(3)} = \alpha_0^{(4)} = 0$.

**Proof** Firstly note that the theorem is trivially provable by direct substitution for the cases $k = 0$ and $k = 1$. We thereafter seek a proof by induction.

Assuming the expression for $B^{k-1}$ is correct, we post-multiply by $ab + cn$ to obtain

$$B^k = (ab)^k + \alpha_k^{(1)} abcnab + \alpha_k^{(2)} abcn(ab)^2 + \alpha_k^{(3)} cn(ab)^2 + \alpha_k^{(4)} cnab + \alpha_k^{(1)} ab(cn)^2 + \alpha_k^{(2)} abnacen + \alpha_k^{(3)} cnaben + \alpha_k^{(4)} (cn)^2 + (ab)^{k-1} cn$$

Since $a \perp b$ and $c \perp n$ it is clear that $ab = -ba$ and $cn = -nc$. Hence $(cn)^2 = -cnnc = -cn^2c = 0$ and $(ab)^2 = -abba = -a^2b^2$ which is scalar and hence commutes with all elements. Further, $a \perp n$ and $b \perp n$ and so $cnaben = cna^2bc = 0$.

Substituting these identities above and simplifying, we obtain

$$B^k = (ab)^k + \alpha_k^{(1)} abcenab + \alpha_k^{(2)} (ab)^2 aben + \alpha_k^{(3)} (ab)^2 cn + \alpha_k^{(4)} cnab + (ab)^{k-1} cn$$

which is of the same form as we assumed for $B^{k-1}$. Equating like coefficients we obtain the required recurrence relations. □
Corollary 1 For \( k \in \mathbb{Z}^+ \),
\[
B^{2k} = (ab)^{2k} + k(ab)^{2k-1}cn + k(ab)^{2k-2}cnab
\]
and
\[
B^{2k+1} = (ab)^{2k+1} + (k + 1)(ab)^{2k}cn + k(ab)^{2k-1}cnab
\]

Proof Starting from \( \alpha_0^{(1)} = 0 \) it is clear that the recurrence relations above imply that \( \alpha_k^{(1)} = \alpha_k^{(2)} = 0 \) \( \forall k \geq 0 \). Substituting \( \alpha_k^{(3)} = \alpha_{k-1}^{(4)} \) it is trivial to show that the relation for \( \alpha_k^{(4)} \) is satisfied by
\[
\alpha_k^{(4)} = \begin{cases} 
\frac{k}{2} (ab)^{k-1} & \text{k even}, \\
\frac{k+1}{2} (ab)^{k-1} & \text{k odd}.
\end{cases}
\]
which, when substituted into the expression for \( B^k \), gives the result stated above. \( \square \)

Theorem 2 If \( B \) is a bivector of the form given in theorem 1 then, defining \( \|ab\|^2 = a^2b^2 = -(ab)^2, c_\parallel \) as the component of \( c \) lying in the plane of \( a \wedge b \) and \( c_\perp = c - c_\parallel \),
\[
\exp(B) = \cos(||ab||) [1 + c_\perp n] + \sinc(||ab||) [ab + c_\parallel n + abc n]
\]

Proof Consider the power series expansion of \( \exp(B) \),
\[
\exp(B) = \sum_{k=0}^{\infty} \frac{B^k}{k!} = \sum_{k=0}^{\infty} \left[ \frac{B^{2k}}{(2k)!} + \frac{B^{2k+1}}{(2k+1)!} \right]
\]
Substituting the expansion for \( B^{2k} \) and \( B^{2k+1} \) from theorem 1, noting that \( (ab)^{2k} = (-1)^k ||ab||^{2k} \), we obtain
\[
\exp(B) = \sum_{k=0}^{\infty} \left( \frac{(-1)^k ||ab||^{2k}}{(2k)!} - \frac{k}{2} \frac{(-1)^k ||ab||^{2k}}{(2k)!} (abc + cnab) \right)
+ \sum_{k=0}^{\infty} \left( \frac{(-1)^k ||ab||^{2k}}{(2k+1)!} (ab + cn) - \frac{k}{2} \frac{(-1)^k ||ab||^{2k}}{(2k+1)!} (abc + cnab - ||ab||^2 cn) \right)
\]
We now substitute the following power-series representations
\[
\cos(z) = \sum_{k=0}^{\infty} \frac{(-1)^k z^{2k}}{(2k)!}
\]
\[
sinc(z) = \sum_{k=0}^{\infty} \frac{(-1)^k z^{2k}}{(2k+1)!}
\]
\[
z \sin(z) = -\sum_{k=0}^{\infty} 2k \frac{(-1)^k z^{2k}}{(2k)!}
\]
\[
\cos(z) - \sin(z) = \sum_{k=0}^{\infty} 2k \frac{(-1)^k z^{2k}}{(2k+1)!}
\]
to obtain
\[
\exp(B) = \cos(||ab||) + \frac{1}{2} \sin(|ab|)(abc + cnab) + \sin(|ab|)(ab + cn)
+ \frac{\sin(|ab|) - \cos(|ab|)}{2 ||ab||^2} (abcab - ||ab||^2 cn)
\]
It is easy to show that \( abc_\perp n = c_\perp nab \) and \( abc_\parallel n = -c_\parallel nab \) and hence
\[
abc + cnab = (abc_\parallel n + abc_\perp n) + (c_\parallel nab + c_\perp nab)
= 2abc_\perp n
\]
and
\[abcnab - \|ab\|^2 cn = ab(c_{\perp} nab + c_{\parallel} nab) - \|ab\|^2 cn = ab(ab_{\perp} n - ab_{\parallel} n) - \|ab\|^2 cn = -\|ab\|^2 (c_{\perp} n - c_{\parallel} n) - \|ab\|^2 cn = -2\|ab\|^2 c_{\perp} n\]
giving
\[\exp(B) = \cos(\|ab\|) + \text{sinc}(\|ab\|)abc_{\perp} n + \text{sinc}(\|ab\|)(ab + cn) - \text{sinc}(\|ab\|)c_{\perp} n + \cos(\|ab\|)c_{\perp} n\]
which, collecting terms in \(\cos(\|ab\|)\) and \(\text{sinc}(\|ab\|)\), is equivalent to
\[\exp(B) = \cos(\|ab\|) [1 + c_{\perp} n] + \text{sinc}(\|ab\|) [ab + cn - c_{\perp} n + abc_{\perp} n]\]
Finally, note that \(cn - c_{\perp} n = c_{\parallel} n\) giving the required expression. \(\square\)

Given some bivector \(B\) in the form \(B = ab + cn\) we can now find its exponentiation. We thus seek the inverse-exponential function \(\ell(R)\)

**Corollary 2** Given the form of \(\exp(B)\) in theorem 2 we can immediately partition it into scalar, bivector and four-vector parts

\[\langle \exp(B) \rangle_0 = \cos(\|ab\|)\]
\[\langle \exp(B) \rangle_2 = \cos(\|ab\|)c_{\perp} n + \text{sinc}(\|ab\|)(ab + c_{\parallel} n)\]
\[\langle \exp(B) \rangle_4 = \text{sinc}(\|ab\|)abc_{\perp} n\]

**Proof** Clear from definition of \(\exp B\) in theorem 2. \(\square\)

**Theorem 3** The inverse-exponential function \(\ell(R)\) is given by
\[\ell(R) = ab + c_{\perp} n + c_{\parallel} n\]

where
\[\|ab\| = \cos^{-1}(\langle R \rangle_0)\]
\[ab = \frac{(\langle R \rangle_2 n) \cdot e}{\text{sinc}(\|ab\|)}\]
\[c_{\perp} n = -\frac{ab \langle R \rangle_4}{\|ab\|^2 \text{sinc}(\|ab\|)}\]
\[c_{\parallel} n = -\frac{ab \langle \langle R \rangle_2 \rangle_2}{\|ab\|^2 \text{sinc}(\|ab\|)}\]

**Proof** It is clear from corollary 2 that the form of \(\|ab\|\) is correct. We thus proceed to show the remaining equations to be true
\[\langle R \rangle_2 = \cos(\|ab\|)c_{\perp} n + \text{sinc}(\|ab\|)(ab + c_{\parallel} n)\]
\[\langle R \rangle_2 n = \text{sinc}(\|ab\|)abn\]
\[(\langle R \rangle_2 n) \cdot e = \text{sinc}(\|ab\|)ab\]
and hence the relation for \(ab\) is correct.
\[\langle R \rangle_4 = \text{sinc}(\|ab\|)abc_{\perp} n\]
\[ab \langle R \rangle_4 = -\|ab\|^2 \text{sinc}(\|ab\|)c_{\perp} n\]
and hence the relation for $c \perp n$ is correct.

\[
\langle R \rangle_2 = \cos(\|ab\|) c \perp n + \text{sinc}(\|ab\|) \left[ ab + c \perp n \right]
\]

\[
ab \langle R \rangle_2 = \cos(\|ab\|) abc \perp n + \text{sinc}(\|ab\|) \left[ abc \parallel n - \|ab\|^2 \right]
\]

\[
\langle ab \rangle \langle R \rangle_2 = \text{sinc}(\|ab\|) abc \parallel n
\]

and hence the relation for $c \parallel n$ is correct. $\square$

We now have a way of mapping a rotor into a linear sub-space and mapping from an element of that space back into the corresponding rotor. We define a rotor interpolation function acting over the set of rotors $R = \{ R_1, R_2, ..., R_n \}$ with interpolation parameter $\lambda$ as

\[
\text{rinterp}(R, \lambda) = \exp \left( \text{interp}(\{ \ell(R_1), \ell(R_2), ..., \ell(R_n) \}, \lambda) \right)
\]

(6.2)

where $\text{interp}()$ is some interpolation function over the bivectors.

We can define easily two standard interpolation functions over the bivectors; linear interpolation,

\[
\text{linterp}(\{ B_1, B_2 \}, \lambda) = \lambda B_1 + (1 - \lambda) B_2
\]

(6.3)

where $\lambda \in (0, 1)$, and quadratic interpolation,

\[
\text{qinterp}(\{ B_1, B_2, B_3 \}, \lambda) = \left[ \frac{B_1 + B_3}{2} - B_2 \right] \lambda^2 + B_2 \lambda + \frac{B_3 - B_1}{2} \lambda
\]

(6.4)

where $\lambda \in (-1, 1)$.

Figure 6.1 shows the result of preforming piecewise-linear and quadratic interpolation of three rotors. Note that smoothness and ‘natural’ nature of the quadratic path.

### 6.3 Comparison of rotor interpolation to other techniques

The SLERP approach naturally requires the partitioning of the interpolation into pure-translation and pure-rotation components. Consequently, it is hard to impose some global constraint or exaggerate desirable properties using SLERP. A particular application may require no rotational component along an axis perpendicular to the path for example. Generating the path and rotation separately during the interpolation makes this hard. Using direct rotor interpolation desirable properties may be specified by placing constraints upon the interpolation path in the generator-space. ‘No-go’ areas can be used to forbid impossible or undesirable positions. For example, forbidding the areas of generator-space corresponding to robot-arm singular points can be used to plan an interpolation which prolongs the life of the arm and ensures smooth real-world movement.

Interestingly, if we force $c = 0$ in the above derivation, we show that our method of rotor interpolation corresponds directly to SLERP. What we have in fact done here is
find a generalisation of the SLERP method. Given the popularity of SLERP, both for its smoothness and efficiency, this bodes well for future applications of this technique.

Mullineux’s approach correctly combines both rotation and translations components of the transform into one multivector. The approach is not obviously generalisable to non-Euclidean problems. The rotor interpolation approach allows for the interpolation of rigid-body transforms in other spaces. For example, path planning in de Sitter space (a common model for the geometry of the Universe) may be useful for investigating the motion and behaviour of astronomical objects.

As of writing, the full relation between path in the bivector generator space and the resulting interpolation is not fully understood. It is anticipated that finding the precise relation will allow for novel methods of optimisation over rotations and interpolation path, perhaps leading to a general method for optimisation over manifolds.
Chapter 7

Future Work

So far a large number of ‘toolkit’-type techniques have been developed. Each has the potential to be useful in a large number of Engineering applications but all need some more development and application to particular problems. This section outlines some possible directions the work could be taken.

7.1 Basic Implementation

\texttt{libcga} is a useful, efficient and compact library which allows for fast implementations of Clifford algebra algorithms. There are, however, areas for improvement.

Recent work [references] on the implementation of computationally intensive algorithms on the \textit{Graphics Processor Unit} (GPU) of modern computer graphics cards using programmable vertex and fragment shaders suggests that the implementation may be greatly improved by the use of the massively-parallel floating point pipelines of these cards. Indeed, the super-exponential growth in computing capability of these devices suggest the resulting implementation may result in lower CPU usage but with higher performance.

7.2 Non-Euclidean Geometries

The basics of manipulating objects within non-Euclidean geometries is now well understood and has already been used to derive some interesting features of de Sitter space, a common model for the late-Universe [are there any references for this?].

What needs to be done now is the application of this to existing problems. Many problems in cosmology require the application of geometry in this space and investigation into whether CGA could be useful in these problems would be a valuable exercise.

It would be best to identify an application which shows the ease at which geometrical

Figure 7.1: A faceted model being translated and rotated in hyperbolic space.
operations can be applied. Figure 7.1 shows a faceted model being moved about the space by various compound rotors. Moving models like this is easy with the approach outlined above but requires complex derivations of rotation and translation operators to achieve the same effect using traditional means.

### 7.3 Rotor Interpolation

This toolkit provides the greatest opportunity for exciting applications. Investigation into the relation between paths in the generator-space and the resulting rotation and path would be useful to develop methods of computing desirable real-world paths by constraining motion in generator-space.

The technique as it stands has obvious applications in computer graphics and robot arm path planning. It has recently been suggested that the path and rotation follows the ‘screw cylinder’ upon which the rotors being interpolated lie. Confirmation of this would be desirable and may allow for novel interpretations of the algorithm in higher dimensions.

It is anticipated that this system will be applied in the later part of the PhD research to a problem in Computer Vision or Computer Graphics. The technique suits itself to the problem of interpolation between three dimensional key frames in animation. One may also use it in skinning, the deformation of some model in response to motion in an associated skeleton.

Perhaps the most useful application is investigating an application of this technique to optimisation over manifolds. We can represent manifolds in some embedding space easily as the set of rotors which take the origin to all points on the manifold and transform the frame so that the manifold is aligned to some axes. We then have a region in the generator space corresponding exactly to the manifold and optimisation of some function within a region of generator space may be an easier problem than optimising over the manifold.

### 7.4 Timetable for Future Work

The following is a proposed timetable for the remainder of the PhD research.
<table>
<thead>
<tr>
<th>Year</th>
<th>Months</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Oct — Dec</td>
<td>Determine relation between paths in the generator space and paths/rotations in the resulting rotor interpolation. Develop methods to promote useful features/obstacle avoidance.</td>
</tr>
<tr>
<td></td>
<td>Jan — Mar</td>
<td></td>
</tr>
<tr>
<td>Year 2</td>
<td>Apr — Jun</td>
<td>Start experimenting with computing using GPU on graphics cards. Determine best strategy and move libcga over to GPU. Benchmark results with original libcga and compare.</td>
</tr>
<tr>
<td></td>
<td>Jul — Sep</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Oct — Dec</td>
<td>Somewhat dependent on progress above. Further the GPU work whilst fully implementing a target application, perhaps in Computer Vision.</td>
</tr>
<tr>
<td></td>
<td>Jan — Mar</td>
<td>Further graphics work. Investigate applications of rotor interpolation in itself. Finish Computer Vision application.</td>
</tr>
<tr>
<td>Year 3</td>
<td>Apr — Jun</td>
<td>Writing up papers &amp; thesis. Present results in a suitable conference. Apply final ‘polish’ to work.</td>
</tr>
<tr>
<td></td>
<td>Jul — Sep</td>
<td></td>
</tr>
</tbody>
</table>
Bibliography


