# An Investigation into Geometric Algebra & it's Implementation in a Ray Tracer

Tim Chauncey
d1459088
bacva 3

# Table of Contents

**Abstract**……..4

**1 Introduction**……..5

**2 3-Dimensional Geometric Algebra**……..6
2.1 Subspaces……6
2.1.1 Bivectors & the Outer Product……..6
2.1.2 Trivectors……..8
2.1.3 Basis Blades……..9
2.1.4 Multivectors……10
2.2 Products……..10
2.2.1 Inner Product……..10
2.2.2 Geometric Product……12
2.2.3 Inner & Outer Product of Multivectors……..14
2.3 Operations……..15
2.3.1 Reverse……...16
2.3.2 Inverse……16
2.3.3 Dual……17
2.3.4 Projection & Rejection……...……17
2.4 Geometry……18
2.4.1 Defining a Line……..19
2.4.2 Defining a Plane……19

**3 Ray Trace Functions**……...20
3.1 Defining a Plane from a Triangle……...20
3.2 Line Plane Intersection……..22
3.3 Finding a Point in a Triangle…….22

**4 Implementation of a Ray Tracer in C++**……..……24
4.1 Multivectors and vectors……24
4.2 Lines, Planes, & Polygons…….24
4.3 Products & Functions…….25
4.4 .Obj Importer……..25
4.5 Creating an Image……..25

**5 Analysis**……..……..26

**6 Conclusion**……...27

**7 Evaluation**……28

**8 References**……29

**Appendix 1 Ray Tracer Algorithm**……………………………………………………..30

**Appendix 2 Using the Ray Tracer**………………………………………………………30

# Abstract

Geometric algebra is an algebra that spans multiple dimensions and provides us with an alternative way to analyse geometry. In this report I provide an introduction to 3-dimensional geometric algebra. I go on to discuss its implementation in a ray tracer so as to determine its relevance within computer graphics.

# 1

# Introduction

Computer graphics has consistently used linear algebra throughout its history. However, many of the papers I read in preparation for this project suggest that geometric algebra is far more diverse.

Through a wide range of research I aim to obtain a general understanding of this branch of mathematics. Using this, I shall attempt to create a C++ library that will allow for geometric algebra computations. I then finally intend to use this library to implement a simple ray tracer. It is through this process that I aim to discover whether 3-dimensional geometric algebra has an appropriate place in computer graphics.

Much of this project has been research based. The majority of the report is an overview of what I have learnt. I start by giving an introduction to geometric algebra in 3-dimensions and introduce the fundamental inner, outer, and geometric products. I then talk about numerous operations and functions that can be derived from these concepts and will aid me in the implementation of a ray tracer. I go on to discuss the methods I have used to code the ray tracer before analysing and concluding my report. In the conclusion, I shall address the matters set out here in the introduction.

# 2

# 3-Dimensional Geometric Algebra

This section of my report outlines the fundamentals of 3-dimensional geometric algebra. Much of this assignment was research based and this portion is a description of what I have learnt.

I would like to note at this point the involvement of Oleg Troy within this assignment. Although our projects had different directions, many sections overlapped greatly. It was with his help that we collectively formed an understanding of the subject at hand. When it came to implementing a C++ library, we continued to aid one another, although each others work is very much his own.

I would also like to take this opportunity to explain that this section of my report is a description of the knowledge I have gained from numerous papers on geometric algebra. Many of the principles and proofs described are derived from other people's work. Where appropriate, I have made references, but I would like to emphasise that any similarities to other papers is purely coincidental.

## 2.1 Subspaces

Geometric algebra introduces the concept of subspaces and is defined through their manipulation.

A scalar is simply a numerical value that has a signed magnitude. These are interpreted as 0-dimensional subspaces. A vector on the other hand is defined by both a magnitude and direction. It occupies a linear portion of 3D space. These are therefore said to represent a 1-dimensional subspace.

With the use of geometric algebra, we are able to find subspace of higher dimensions.

### *2.1.1 Bivectors & the Outer Product*

Geometric algebra defines an operator called the outer product which is denoted by the ^ (wedge) symbol. In terms of vectors, this essentially has the effect of extruding one vector along another. The result is a 2-dimensional subspaces know as a bivector. A bivector has an area and an orientation, but its shape is undefined. Figure 2.1 demonstrates the construction of this 2D subspace.

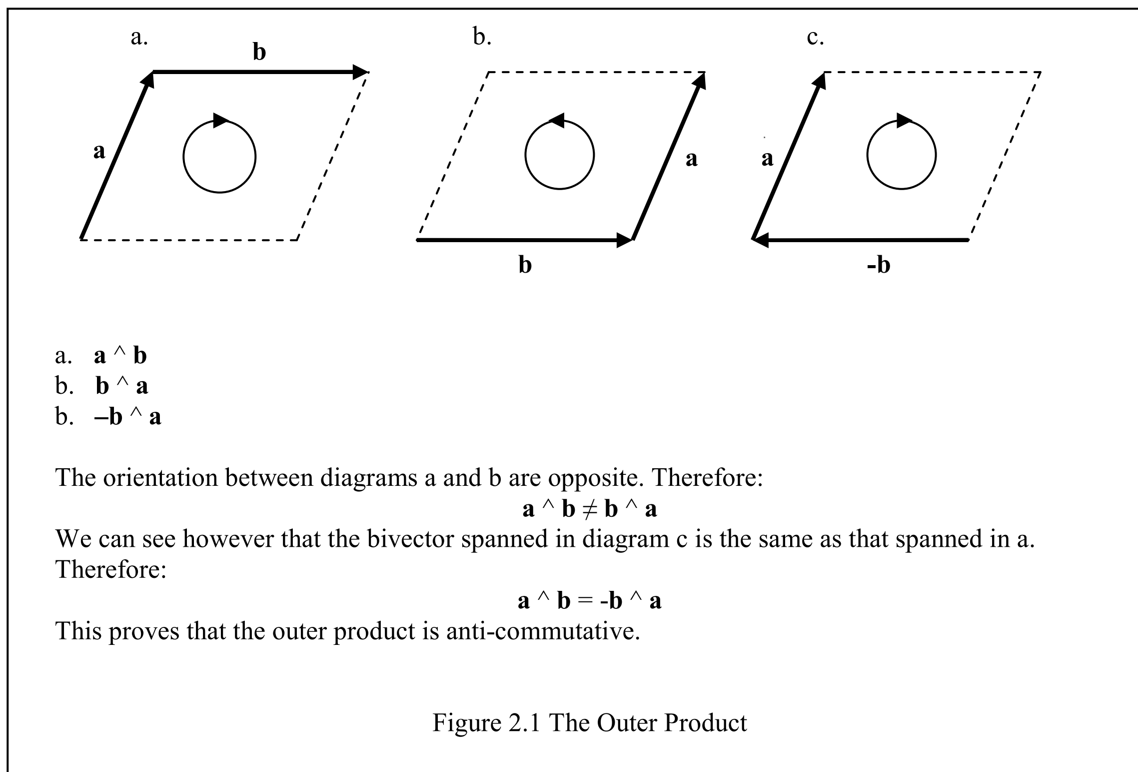



Figure 2.1 The Outer Product

We have now defined a 2-dimensional subspace by extruding a 1-dimensional subspace along another 1-dimensional subspace.
From figure 2.1, we can see that the outer product is anti-commutative. That is to say:

$$\mathbf{a} \wedge \mathbf{b} = -\mathbf{b} \wedge \mathbf{a} \qquad (2.1)$$

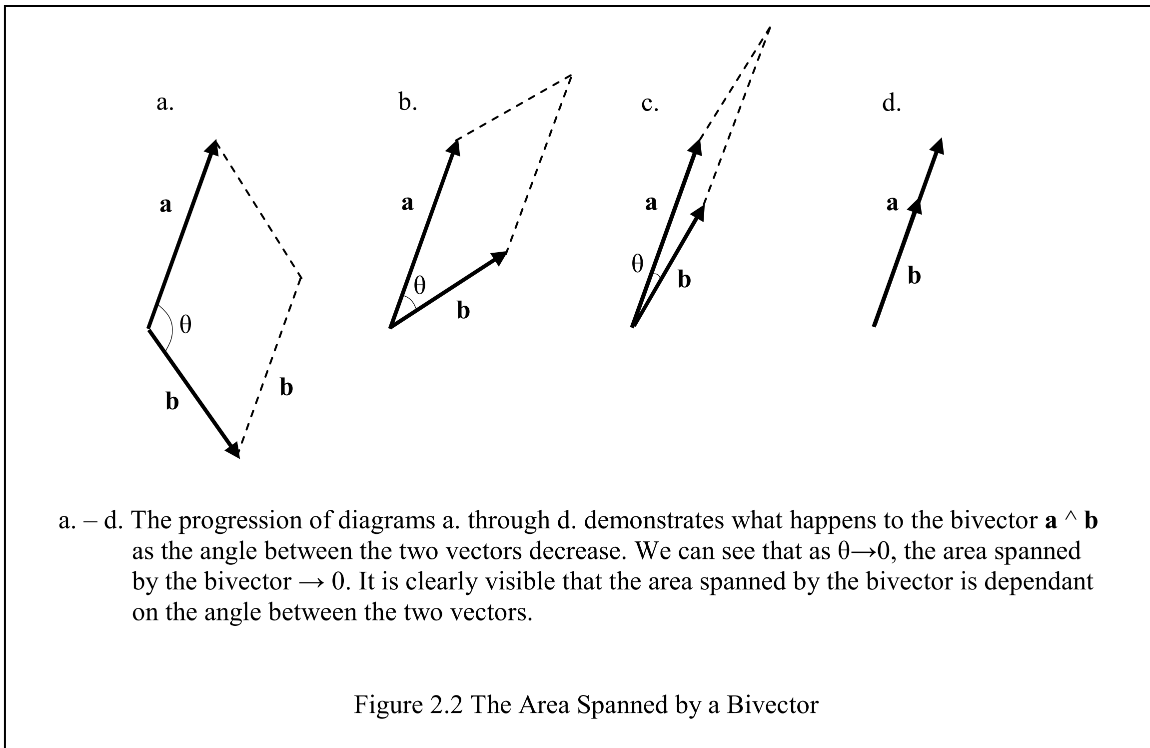
Some other properties of the outer product are defined by Jaap Suter [1] :

$(\lambda \mathbf{a}) \wedge \mathbf{b} = \lambda (\mathbf{a} \wedge \mathbf{b})$ associative scalar multiplication() (2.2)

$\lambda (\mathbf{a} \wedge \mathbf{b}) = (\mathbf{a} \wedge \mathbf{b}) \lambda$ commutative scalar multiplication (2.3)

$\mathbf{a} \wedge (\mathbf{b} + \mathbf{c}) = (\mathbf{a} \wedge \mathbf{b}) + (\mathbf{b} \wedge \mathbf{c})$ distributive over vector addition (2.4)

It is important to note that the outer product between two vectors behaves much like the cross product. Whereas the size of the resulting vector of the cross product is dependant upon the angle between the two vectors, so is the magnitude of their bivector. Also, the cross product of two parallel vectors is equal to 0, as is the magnitude of the bivector that spans them. This can be clearly seen in figure 2.2.

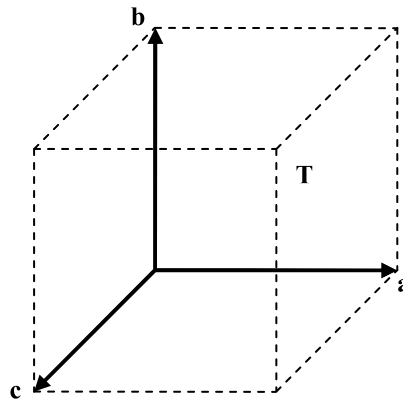



a. – d. The progression of diagrams a. through d. demonstrates what happens to the bivector **a** ^ **b** as the angle between the two vectors decrease. We can see that as $\theta \rightarrow 0$, the area spanned by the bivector → 0. It is clearly visible that the area spanned by the bivector is dependant on the angle between the two vectors.

Figure 2.2 The Area Spanned by a Bivector

### *2.1.2 Trivectors*

Consider three linearly dependant vectors **a,b,** and **c**. **a**^**b** would result in a bivector AB. If we were to perform the cross product between AB and **c**, we would be extruding a bivector along a vector. The result is a 3-dimensional subspace know as a trivector as depicted in figure 2.3.




The result of the outer product between three vectors, **a**, **b**, and **c** is the spanned volume, or trivector **T**.

**T = a ^ b ^ c**

Figure 2.3 Trivector

A trivector is essentially an oriented volume. It has a magnitude and a volume but, as with bivectors, it has no specific shape. The cube used to depict it in figure 2.3 is merely for visualisation purposes.

### *2.1.3 Basis Blades*

It is possible to define subspaces of higher dimensions. For example, the outer product between a vector and a trivector would result in a 4-dimensional subspace. However, since we are only interested in 3-dimensional geometric algebra, a trivector is the subspace of the highest order we can identify.

Now that we have defined what subspaces are, we need a way to represent them. The method for this is very similar to describing a vector in linear algebra where you would use the base vectors x,y, and z to define a vector's components.

Geometric algebra uses the term *k*-blade to describe a subspace, where *k* is the dimension spanned by that subspace. A scalar spans 0-dimensions and so is called a 0-blade; a vector spans 1-dimension and so is called a 1-blade; a bivector spans 2-dimensions and so is called a 2-blade; a trivector spans 3-dimensions and so is called a 3-blade.

In 3D Euclidean space, we define a set of basis blades that we use to describe arbitrary blades. There is 1 base 0-blade, 3 base 1-blades, 3 base 2-blades, and 1-base 3-blade.

The 0-blade is a scalar. The three base 1-blades are simply three base vectors similar to **x**,**y**, and **z** in linear algebra. However, in geometric algebra these are called **e1**,**e2**, and **e3** respectively. Since we have three base 1-blades, we are able to create three base bivectors. These are **e1^e2**, **e2^e3**, and **e3^e1**. The single base 3-blade is defined by **e1^e2^e3.**

We are now able to define any scalar, vector, bivector, or tirvector in 3D Euclidean space through a combination of these basis blades.

### *2.1.4 Multivectors*

The general entity of geometric algebra that is used for calculations is called the multivector. Whereas a vector in linear algebra is a combination of the basis vectors, a multivector in geometric algebra is the combination of the basis blades. A multivector **A** can be defined as:

$$\mathbf{A} = \alpha_1 + \alpha_2\mathbf{e1} + \alpha_3\mathbf{e2} + \alpha_4\mathbf{e3} + \alpha_5\mathbf{e1{\wedge}e2} + \alpha_6\mathbf{e2{\wedge}e3} + \alpha_7\mathbf{e3{\wedge}e1} + \alpha_8\mathbf{e1{\wedge}e2{\wedge}e3} \qquad (2.5)$$

We treat multivectors in a similar way to complex numbers. We are clearly unable to add a vector to a bivector, or any other element of a different dimension. We therefore keep each component separate and deal with them individually.

## 2.2 Products

### *2.2.1 Inner Product*

We have already stated that the outer product is an extension of the cross product. Similarly, the inner product can be described as an extension of dot product and is denoted by the · symbol.

The inner product is a measure of perpendicularity [2]. The value of the dot product in linear algebra is dependant upon the angle between two vectors. If the vectors are perpendicular, the dot product is equal to 0. If they are parallel, the dot product is 1. The inner product allows us to perform such computations on blades that span multiple dimensions.

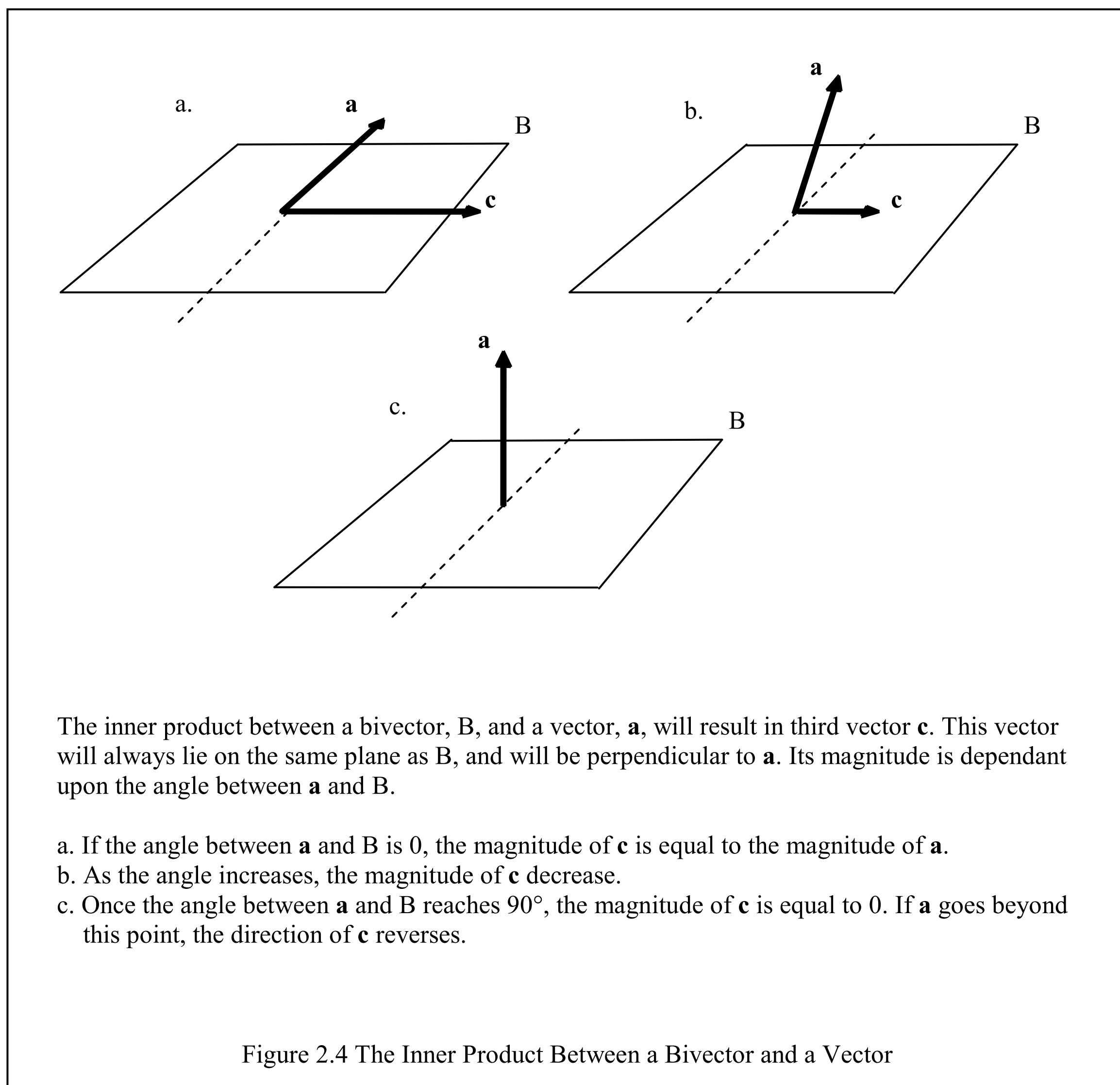
The result of the inner product between arbitrary blades is almost the opposite of the outer product. Whereas the outer product spans subspaces, the inner product deconstructs them, i.e. it subtracts the degrees of subspaces. For example,
(2-blade) · (1-blade) = (1-blade), (2-1 = 1).

The easiest way to describe this product is to demonstrate its use between various blades.

The inner product between two scalars is simply the scalar product.

The inner product between two blades of the same degree is a scalar. This is simple to comprehend if we subtract the degrees of subspaces. The magnitude of the scalar is a measure of the perpendicularity between the two subspaces.

I shall now demonstrate the inner product between a bivector A and a vector **b**. If we subtract a 1-blade, **b**, from a 2-blade, A, the result will be a 1-blade, in this case a vector **c**. This vector will lie on the plane defined by A and will be perpendicular to **b**. As with the dot product, the angle between A and **b** will determine the magnitude of **c**. If **b** is orthogonal to A, then the magnitude of **c** is 0. If **b** lies on the same plane as A, the magnitude of **c** is equal to the magnitude of **b**. The inner product between a bivector and a vector is a measure of the perpenicularity between the two. The above paragraph is demonstrated further in figure 2.4.




The inner product between a bivector, B, and a vector, **a**, will result in third vector **c**. This vector will always lie on the same plane as B, and will be perpendicular to **a**. Its magnitude is dependant upon the angle between **a** and B.

a. If the angle between **a** and B is 0, the magnitude of **c** is equal to the magnitude of **a**.
b. As the angle increases, the magnitude of **c** decrease.
c. Once the angle between **a** and B reaches 90°, the magnitude of **c** is equal to 0. If **a** goes beyond this point, the direction of **c** reverses.

Figure 2.4 The Inner Product Between a Bivector and a Vector

I shall finally consider the inner product between a trivector **A** and a bivector B as this has a particularly useful outcome. We know that if we subtract the degree of dimensionality of the bivector from the trivector, the result will be a vector. Since we cannot define how perpendicular a plane is to a volume, the magnitude of this vector remains constant no matter how the bivector is oriented. However, its direction is always pendindicular to the bivector. Thus, if we wish to find the normal to a bivector, we simply find its inner product with the base trivector.

### *2.2.2 Geometric Product*

The geometric product is a combination of both the outer and inner products and forms the basis for geometric algebra computations.

For two vectors **a** and **b**, the geometric product is defined as:

$$\mathbf{ab} = \mathbf{a} \wedge \mathbf{b} + \mathbf{a} \cdot \mathbf{b} \qquad (2.6)$$

So far, we have only discussed the inner and outer product, and now the geometric product in terms of vectors. However, we can use these definitions to extend our geometric product for use on multivectors. If we are able to find the geometric product of the basis blades, we will be able to calculate the geometric product between any two multivectors. From this, we can also derive the inner and outer products for multivectors.

Some general properties of the geometric product have been defined by Jaap Suter [3]:

| | | |
|---|---|---|
| $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$ | associativity | (2.7) |
| $\lambda\mathbf{A} = \mathbf{A}\lambda$ | commutative scalar multiplication | (2.8) |
| $\mathbf{A}(\mathbf{B} + \mathbf{C}) = \mathbf{AB} + \mathbf{AC}$ | distributive over addition | (2.9) |

We have previously defined our 3D space in terms of the three orthogonal base vectors **e1**, **e2**, and **e3**. Further base subspaces are also defined in terms of the outer product of these vectors.

Let us firstly consider the geometric product of the basis vector **e1** with itself:

$$\mathbf{e1e1} = \mathbf{e1} \wedge \mathbf{e1} + \mathbf{e1} \cdot \mathbf{e1}$$

We already know that the outer product between two parallel vectors is equal to 0. We are therefore able to disregard the **e1** ^ **e1** portion of this equation. We also know that the inner product of a vector with itself is equal to 1.

$$\therefore\ \mathbf{e1e1} = 0 + 1$$
$$= 1 \qquad (2.10)$$

Let us now consider the geometric product between the two base vectors **e1** and **e2**:

$$\mathbf{e1e2} = \mathbf{e1} \wedge \mathbf{e2} + \mathbf{e1} \cdot \mathbf{e2}$$

Since **e1** and **e2** are orthogonal, we know that the inner product between them is equal to 0. We are therefore able to remove the **e1 · e2** portion of the equation

$$\therefore\ \mathbf{e1e2} = \mathbf{e1} \wedge \mathbf{e2} + 0$$
$$\mathbf{e1e2} = \mathbf{e1} \wedge \mathbf{e2} \qquad (2.11)$$

So the outer product between two base vectors can be expressed as their geometric product.

If we reverse the order of these two base vectors we obtain:

$$\begin{aligned}\mathbf{e2e1} &= \mathbf{e2} \wedge \mathbf{e1} + \mathbf{e2} \cdot \mathbf{e1} \\ &= \mathbf{e2} \wedge \mathbf{e1} + 0 \\ &= -\,\mathbf{e1} \wedge \mathbf{e2} \qquad \text{(from equation (2.1))} \\ &= -\,\mathbf{e1e2} \qquad (2.12)\end{aligned}$$

So, if we flip the order of the geometric product, we negate the result.

The equations (2.7), (2.10), (2.11), and (2.12) provide us with four rules that allow us to simplify the geometric product between any of the basis blades. I will demonstrate this in one further example below.

Consider the base bivector **e1^e2** and the base trivector, **e1^e2^e3**. (2.11) allows us to express their geometric product in the following manner:

$$(\mathbf{e1} \wedge \mathbf{e2})(\mathbf{e1} \wedge \mathbf{e2} \wedge \mathbf{e3}) = (\mathbf{e1\ e2})\ (\mathbf{e1\ e2\ e3})$$

Using equation (2.7) we can write this without the brackets as we are able to group any combination of vectors:

$$(\mathbf{e1} \wedge \mathbf{e2})\ (\mathbf{e1} \wedge \mathbf{e2} \wedge \mathbf{e3}) = \mathbf{e1\ e2e1\ e2\ e3}$$

Now, using (2.12) we can flip the **e2** and **e1** terms in the centre as long as we negate the result:

$$(\mathbf{e1} \wedge \mathbf{e2})\ (\mathbf{e1} \wedge \mathbf{e2} \wedge \mathbf{e3}) = -\,\mathbf{e1\ e1e2\ e2\ e3}$$

From equation (2.10) we know that the **e1e1** and **e2e2** portions are both equal to 1. We can therefore conclude that:

$$(\mathbf{e1} \wedge \mathbf{e2})\ (\mathbf{e1} \wedge \mathbf{e2} \wedge \mathbf{e3}) = -\,\mathbf{e3}$$

Now that I have demonstrated the method to calculate the geometric product between any of the basis blades, we are able to construct a multiplication table that we can refer to in order to compute the geometric product between two arbitrary multivectors. This is shown in figure 2.5.

| | s | e1 | e2 | e3 | e12 | e23 | e31 | e123 |
|---|---|---|---|---|---|---|---|---|
| **s** | s | e1 | e2 | e3 | e12 | e23 | e31 | e123 |
| **e1** | e1 | s | (-e12) | e31 | (-e2) | e123 | e3 | e23 |
| **e2** | e2 | e12 | s | (-e23) | e1 | (-e3) | e123 | e31 |
| **e3** | e3 | (-e31) | e23 | s | e123 | e2 | (-e1) | e12 |
| **e12** | e12 | e2 | (-e1) | e123 | (-s) | e31 | (-e23) | (-e3) |
| **e23** | e23 | e123 | e3 | (-e2) | (-e31) | (-s) | e12 | (-e1) |
| **e31** | e31 | (-e3) | e123 | e1 | e23 | (-e12) | (-s) | (-e2) |
| **e123** | e123 | e23 | e31 | e12 | (-e3) | (-e1) | (-e2) | (-s) |

Figure 2.5 Multiplication Table for the Geometric Product between Two Multivectors

### *2.2.3 Inner & Outer Products of Multivectors*

Since the geometric product is the sum of the inner and outer products, we can use a few rules to isolate the sections of the table in figure 2.5 that refer to each product. Thus we will be able to obtain a multiplication table for the inner and outer product of multivectors.

Let us firstly consider the rules for the outer product:

1. If two terms contains vectors that have components that are independent of one another, the outer product between them cannot be simplified and the result is included. For example, **e1** ^ **(e2** ^ **e3)** cannot be broken down further.

2. If two terms contain vectors that have similar components, the result of the outer product will be 0. For example, **e1** ^ (**e1** ^ **e2**) = 0. The two terms here are **e1** and **e1** ^ **e2**. We can see that **e1** is common to both terms. Therefore, since **e1** ^ **e1** = 0, the term must result to 0.

Using these two rules we can construct the multiplication table for the outer product as in figure 2.6.

|  | s | e1 | e2 | e3 | e12 | e23 | e31 | e123 |
|---|---|---|---|---|---|---|---|---|
| **s** | s | e1 | e2 | e3 | e12 | e23 | e31 | e123 |
| **e1** | e1 | 0 | (-e12) | e31 | 0 | e123 | 0 | 0 |
| **e2** | e2 | e12 | 0 | (-e23) | 0 | 0 | e123 | 0 |
| **e3** | e3 | (-e31) | e23 | 0 | e123 | 0 | 0 | 0 |
| **e12** | e12 | 0 | 0 | e123 | 0 | 0 | 0 | 0 |
| **e23** | e23 | e123 | 0 | 0 | 0 | 0 | 0 | 0 |
| **e31** | e31 | 0 | e123 | 0 | 0 | 0 | 0 | 0 |
| **e123** | e123 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Figure 2.6 Multiplication Table for the Outer Product between Two Multivectors

We can use a similar technique to derive a multiplication table for the inner product. The rules in this case are as follows:

1. If two terms have components that are independent of one another, e.g. **e1e2** and **e3**, the inner product between them will equal 0 and so the result is disregarded. We know this must be the case since the inner product between two perpendicular vectors is equal to 0. Therefore, the result must compute to 0.

2. If two terms contain similar components, the inner product between these is equal to 1 and so the result is included. For example, **e1** · (**e1** ^ **e2**) will be included as both terms contain an **e1** component and **e1** · **e1 = 1.**

The multiplication table for the inner product can be seen in figure 2.7.

|  | s | e1 | e2 | e3 | e12 | e23 | e31 | e123 |
|---|---|---|---|---|---|---|---|---|
| **s** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **e1** | 0 | s | 0 | 0 | (-e2) | 0 | e3 | e23 |
| **e2** | 0 | 0 | s | 0 | e1 | (-e3) | 0 | e31 |
| **e3** | 0 | 0 | 0 | s | 0 | e2 | (-e1) | e12 |
| **e12** | 0 | e2 | (-e1) | 0 | (-s) | 0 | 0 | (-e3) |
| **e23** | 0 | 0 | e3 | (-e2) | 0 | (-s) | 0 | (-e1) |
| **e31** | 0 | (-e3) | 0 | e1 | 0 | 0 | (-s) | (-e2) |
| **e123** | 0 | e23 | e31 | e12 | (-e3) | (-e1) | (-e2) | (-s) |

Figure 2.7 Multiplication Table for the Inner Product between Two Multivectors

## 2.3 Operations

We have defined the three major products of geometric algebra between any two arbitrary multivectors. I will now use these to describe a number of operations we can perform that will aid us at the implementation of a ray tracer. Although there are a variety of functions that can be shown, I shall only be describing those that will be of use to me.

### *2.3.1 Reverse*

The reverse of a multivector is defined by switching the geometric products of its basis blades. So, **e1e2** would become **e2e1**, **e2e3** would become **e3e2**, **e3e1** would become **e1e3**, and **e1e2e3** would become **e3e2e1**. This has the effect of changing the sign of each of the bivectors and the trivector, as shown below:

**e2e1** = - **e1e2**

**e3e2** = - **e2e3**

**e1e3** = - **e3e1**

**e3e2e1** = - **e3e1e2**
= **e1e3e2**
= - **e1e2e3**

The reverse is denoted by $^\dagger$, and we can see its effect on a multivector A:

$$\mathbf{A} = \alpha_1 + \alpha_2\mathbf{e1} + \alpha_3\mathbf{e2} + \alpha_4\mathbf{e3} + \alpha_5\mathbf{e1e2} + \alpha_6\mathbf{e2e3} + \alpha_7\mathbf{e3e1} + \alpha_8\mathbf{e1e2e3}$$

$$\mathbf{A}^\dagger = \alpha_1 + \alpha_2\mathbf{e1} + \alpha_3\mathbf{e2} + \alpha_4\mathbf{e3} - \alpha_5\mathbf{e1e2} - \alpha_6\mathbf{e2e3} - \alpha_7\mathbf{e3e1} - \alpha_8\mathbf{e1e2e3}$$

### *2.3.2 Inverse*

Geometric algebra can provide a particularly powerful operation in that we are able to find the inverse of multivectors. This allows us to divide by multivectors, an operation that we cannot perform in linear algebra.

Not all multivectors have an inverse. Fortunately, there are a particular set which do, known as versors. A versor is the geometric product of vectors. This is particularly useful when working in computer graphics as most of the multivectors we will be using will in fact take the form of vectors.

The inverse of a versor is defined by:

$$A^{-1} = \frac{A^\dagger}{A^\dagger A}$$

The important thing to understand here is that $A^\dagger A$, will always produce a scalar that is equal to the sum of the square of the coefficients of the components:

$$A^\dagger A = \sum_{k=1}^{k=n} a_k$$

where *a* is the coefficient of the component.

As a result we can prove that the equation for the inverse is correct by multiplying both sides b A:

$$A^{-1}A = \frac{A^{\dagger}A}{A^{\dagger}A}$$
$$= 1$$

### 2.3.3 *Dual*

The dual of a multivector in 3-dimensions is simply its geometric product with the inverse of the base trivector. It is denoted by *. The base trivector is given by **I**. This is essentially a multivector whose coefficients are all 0 except for the coefficient of **e1e2e3,** i.e. **I** = 0 + 0**e1** + 0**e2** + 0**e3** + 0**e1e2** + 0**e2e3** + 0**e3e1** + 1**e1e2e3.**

$$\mathbf{A}^* = \mathbf{A}\mathbf{I}^{-1}$$

When defining the inner product between a bivector and a trivector, I stated that the result would be the vector normal to the bivector. This is in fact the negative normal to the bivector, as it points in the opposite direction to the bivector's orientation.

Using the dual solves this problem. If **A** is a bivector, the dual will return the normal to **A** in the correct direction, hence the *inverse* of the trivector **I**.

### 2.3.4 *Projection & Rejection[4]*

Consider a vector **a** and a bivector B. The projection will return the projected vector of **a** onto B. It will be the component of **a** that lies on the same plane as the bivector. This is represented by $\mathbf{a}_{||}$.

The rejection will return the component of **a** that is perpendicular to B. This is denoted by $\mathbf{a}_{\perp}$.

The equations for the projection is given by:

$$\mathbf{a}_{||B} = (a \cdot B)B^{-1}$$

We can prove this since we know that outer product between $\mathbf{a}_{||B}$ and B will be 0. This has to be the case since $\mathbf{a}_{||B}$ lies on the same plane as B. We also know that the inner product between $\mathbf{a}_{\perp B}$ and B will be 0 since $\mathbf{a}_{\perp B}$ is perpendicular to B.

$$\mathbf{a}_{||B} \wedge B = 0 \qquad (2.13)$$

$$\mathbf{a}_{\perp B} \cdot B = 0 \qquad (2.14)$$

If we perform the geometric product between $\mathbf{a}_{||B}$ and B we get:

$$\mathbf{a}_{||B}\, B = \mathbf{a}_{||B} \wedge B + \mathbf{a}_{||B} \cdot B$$

$$\mathbf{a}_{\|B}\, B = \mathbf{a}_{\|B} \cdot B$$

We can now add $\mathbf{a}_{\perp B} \cdot B$ to the right hand side of the equation as this equals 0:

$$\mathbf{a}_{\|B}\, B = \mathbf{a}_{\|B} \cdot B + \mathbf{a}_{\perp B} \cdot B$$

$$\mathbf{a}_{\|B}\, B = (\mathbf{a}_{\|B} + \mathbf{a}_{\perp B}) \cdot B$$

$$\mathbf{a}_{\|B}\, B = \mathbf{a} \cdot B$$

$$\mathbf{a}_{\|B} \quad = (a \cdot B)B^{-1}$$

The equations for the projection is given by:

$$\mathbf{a}_{\perp B} = (\mathbf{a} \wedge B)B^{-1}$$

This can be proven in a similar manner:

$$\mathbf{a}_{\perp B}\, B = \mathbf{a}_{\perp B} \wedge B + \mathbf{a}_{\perp B} \cdot B$$

$$\mathbf{a}_{\perp B}\, B = \mathbf{a}_{\perp B} \wedge B$$

Add $\mathbf{a}_{\|B} \wedge B$ to the right hand side of the equation gives:

$$\mathbf{a}_{\perp B}\, B = \mathbf{a}_{\perp B} \wedge B + \mathbf{a}_{\|B} \wedge B$$

$$\mathbf{a}_{\perp B}\, B = (\mathbf{a}_{\perp B} + \mathbf{a}_{\|B}) \wedge B$$

$$\mathbf{a}_{\perp B}\, B = \mathbf{a} \wedge B$$

$$\mathbf{a}_{\perp B} = (\mathbf{a} \wedge B)B^{-1}$$

## 2.4 Geometry

In order to produce a ray tracer, there are two fundamental pieces of geometry that we have to describe. These are a line and a plane. Here, I will demonstrate how we would go about doing this.

### *2.1.4 Defining a Line*

Since I am only working in 3-dimensions, I am able to define a line in the same way as linear algebra. It is described by a position vector **p**, which is a point through which the line passes, and a direction vector **v** to describe the direction of the line. The only difference, is that these two vectors are essentially multivectors with coefficients of its 1-blades only.

### *2.1.5 Defining a Plane*

An arbitrary plane can be defined in 3-dimensional geometric algebra through a bivector, B, and a scalar, d. B describes the orientation of the plane whilst d gives its perpendicular distance from the origin. Since a plane is infinitely large, it cannot be translated in any other direction other than perpendicular to the origin. A scalar value can therefore define its position in space.

# 3

# Ray trace Functions

Appendix 2 provides the basic pseudo-code for the ray tracer that I have attempted to implement. It shows particular functions that I have had to create to perform specific tasks. In this section I will discuss what these functions are and how they have been implemented.

## 3.1 Defining a Plane from a Polygon

In order to find if a ray intersects with a polygon, I first check to see if that ray intersects the plane on which the polygon lies. If it does, I can then check to see if this point of intersection is within the polygon, and thus decide whether to render the pixel. This suggests that each polygon must have a specifically defined plane with which to check for intersections. Since I am importing objects as .obj files, these planes are not pre-defined. I have therefore established a method for defining a plane from the points of the polygon.

In order to simplify this method, I have decided to use a triangulated mesh. This will provide me with three position vectors, **a**, **b,** and **c**. with which to establish a plane.

The first step in this procedure is to translate the triangle so that point **a** is at the origin as in figure 4.1a.

$$\mathbf{a}` = \mathbf{a} - \mathbf{a}$$
$$\mathbf{b}` = \mathbf{b} - \mathbf{a}$$
$$\mathbf{c}` = \mathbf{c} - \mathbf{a}$$

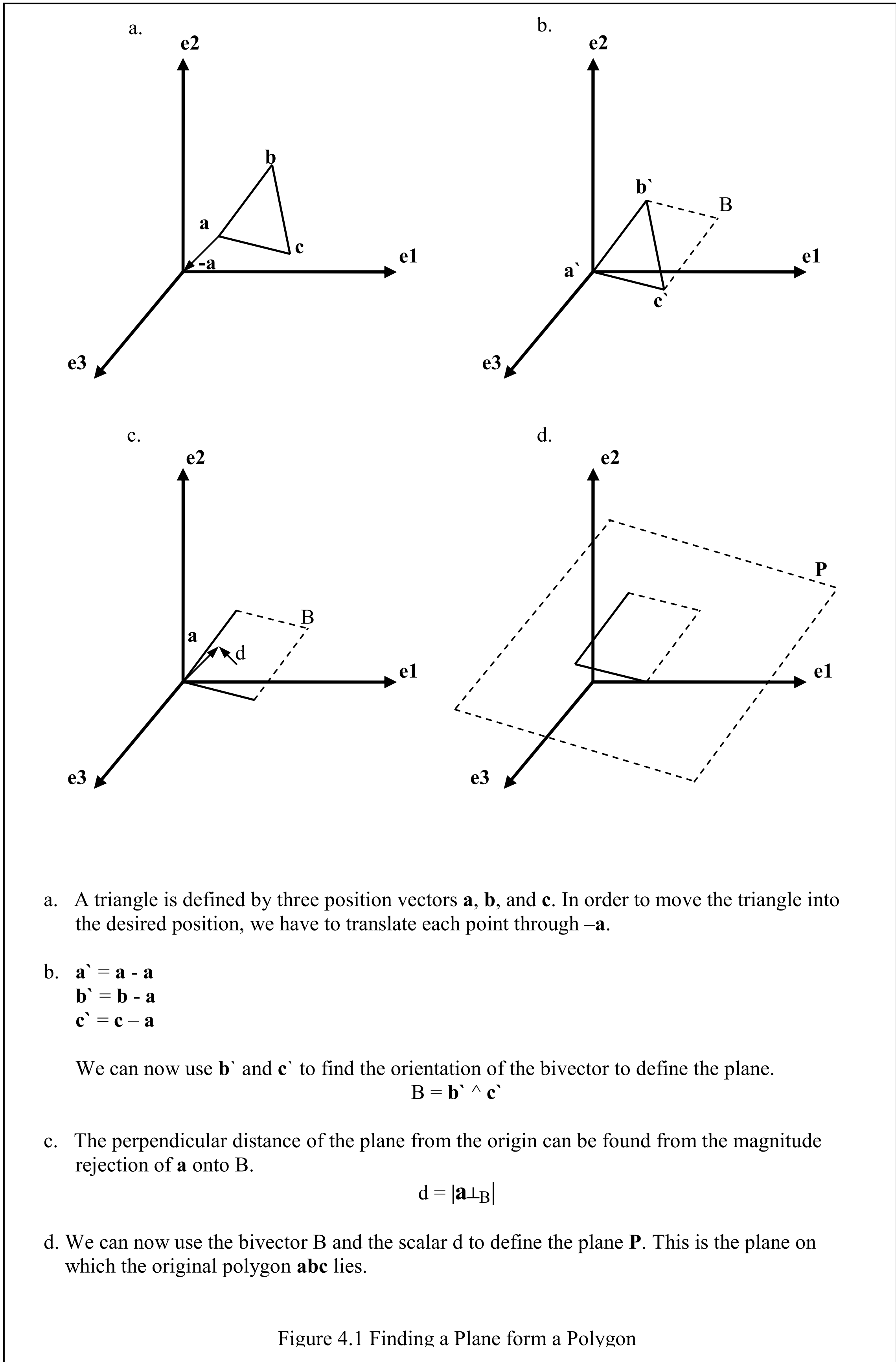
We can now describe the orientation of the plane by calculating the outer product between **b`** and **c`**. This will result in a bivector B as in figure 4.1b.

$$B = \mathbf{a}` \wedge \mathbf{c}`$$

Now that we have orientation of the plane, we need to find its perpendicular distance from the origin.

In order to get the triangle from its position to the origin, we had to translate it through a vector –**a**. We therefore know that to get it back to its original position, we would have to translate it through **a**. In the previous section, I defined the rejection operation. By finding the rejection of **a** on B, we will find the component of **a** that is perpendicular to B. The magnitude of this vector will give the distance, d, that the plane is from the origin. This is shown in figure 4.1c.

$$d = |\mathbf{a}_{\perp B}|$$




a. A triangle is defined by three position vectors **a**, **b**, and **c**. In order to move the triangle into the desired position, we have to translate each point through –**a**.

b. **a`** = **a** - **a**
**b`** = **b** - **a**
**c`** = **c** – **a**

We can now use **b`** and **c`** to find the orientation of the bivector to define the plane.

$$B = \mathbf{b`} \wedge \mathbf{c`}$$

c. The perpendicular distance of the plane from the origin can be found from the magnitude rejection of **a** onto B.

$$d = |\mathbf{a}_{\perp B}|$$

d. We can now use the bivector B and the scalar d to define the plane **P**. This is the plane on which the original polygon **abc** lies.

Figure 4.1 Finding a Plane form a Polygon

## 3.2 Line Plane Intersection*[5]*

Now that we have defined a plane for each polygon in the scene, we have to check whether a ray actually intersects that plane.

Consider a line, l, and a plane, p. **a** is the point through which the line passes and **u** is its direction. B is the bivector of the plane and d is its distance from the origin. The point $\mathbf{p_i}$ of intersection of of l and p is given by the following equation:

$$\mathbf{p_i} = \mathbf{a} - \frac{((\mathbf{a} \wedge B)^* - d)\mathbf{u}}{(\mathbf{u} \wedge B)^*}$$

## 3.3 Finding a Point in a Triangle

If a ray intersects a plane, we need to establish whether the point of intersection is inside the triangle that lies on that plane. The method that I have used is only defined for a triangle. However, the principles could be applied to a polygon of any shape. It is a commonly used algorithm in linear algebra. However, with the aid of a fellow student, Oleg Troy, we were able to convert it to work using geometric algebra.

To find whether a particular point is inside a triangle we perform three checks. **a**, **b**, and **c** are the shapes vertices. Firstly, we see whether point, **p**, is on the correct side of **ab** for it to be inside the triangle. If it is, we do the same check but this time with edge **bc**. If the point proves to be on the right side of **bc**, we finally check with edge **ac**. If all three checks are true, then the point must be within the triangle. This is further described in figure 4.2.

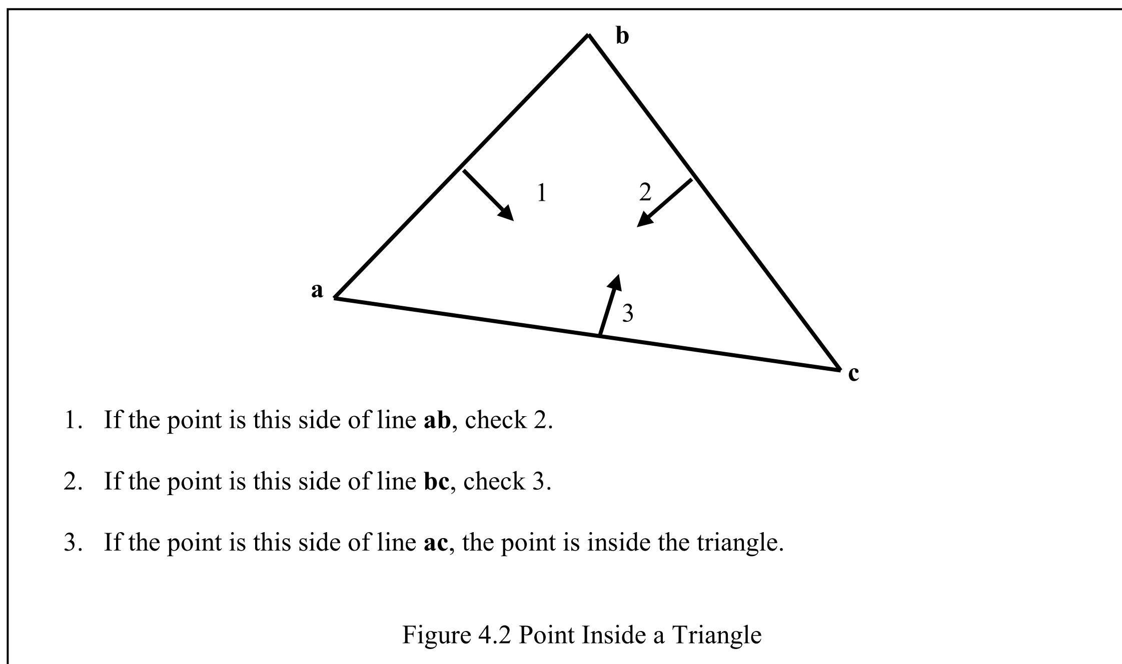



Figure 4.2 Point Inside a Triangle

To determine whether a point is on the correct side of an edge we perform two outer products. Lets consider edge **ab** and point **p**. We would first compute the outer product between **ab** and **ap** to get bivector A. We would then compute the outer product of **ab** with **ac**, to get bivector B.

$$A = \mathbf{ab} \wedge \mathbf{ap}$$
$$B = \mathbf{ab} \wedge \mathbf{ac}$$

By comparing the orientation of the two bivectors, we can tell whether or not **p** is on the correct side of **ab** for it to be inside the triangle. We can compare them using the inner product to obtain a scalar, s.

$$s = A \cdot B$$

If s is positive, the orientation of both bivectors are the same, i.e. the orientations are either both positive or both negative. **p** therefore lies on the correct side of the edge. If s is negative, one of the orientations is different to the other. **p** must lie on the wrong side of the edge and is thus not in the triangle.

# 4

# Implementation of a Ray tracer in C++

So far I have discussed the theory behind geometric algebra and the mathematical solutions to the main functions that I will require for a ray tracer. In this section, I will talk about how I have implemented this in C++.

## 4.1 Multivectors & Vectors

In order to represent a multivector, I have created a class that stores 8 doubles, one to represent each coefficient of the basis blades.

It is clear, however, that not all of these elements shall be used the majority of the time. This is especially so in computer graphics when we tend to be dealing mostly with vectors. I have therefore also created a vector class that stores only three doubles. These correspond to the 1-blade components of a multivector, i.e. its vector components.

Both the multivector and vector classes contain methods for setting and retrieving the data within the class, and also default and user defined constructors.

## 4.2 Lines, Planes, & Polygons

I have created three other classes to store primitive geometry.

A line is a class that holds two vectors. The first is a position vector, p, for the point through which the line passes. The second is a direction vector, direction. There are two other vectors in the class named x1 and x2. These are used for the purpose of drawing the line in openGL.

A plane class holds a multivector called bivector, and a scalar called offset. The multivector should only have values in its 2-blade components as these are what define the orientation of the bivector. Offset is the magnitude of the perpendicular distance that the plane is away from the origin.

I have designed my program only to use triangular polygons. I was therefore able to create a polygon class that stores an array of three vectors. These hold the three position vectors that define the points of the triangle.

Each of these classes contains methods for setting and retrieving the data in the class. They also all have a default and user defined constructor, and a print method that will allow the private variables to be printed to the screen. The line and polygon classes contain a draw method that will draw them in openGL.

## 4.3 Products & Functions

I have implemented all the products described in this paper in a header file called products.h. I have included multiple versions of the products as they have been overloaded to allow for their use between any combination of multivectors and vectors.

All of the operations described, along with some others such as rotations, reflections, and line intersections, are implemented in operations.h. These use the functions from products.h to implement the maths behind the operations described previously.

## 4.4 .Obj importer

I have had to implement my own basic importer for objects stored in .obj files. My program specifically works for the primitive geometry that I have described. I therefore need to store the mesh from the .obj file in terms of the vector and polygon objects I have defined.

At the beginning of the program I create an array of vectors called vertex, and an array of polygons called polys. It is in these arrays that the vertices and triangular polygons of the .obj files are stored.

## 4.5 Creating an image

To create the outputted image, I used the Magick++.h[6] header file from www.imagemagick.org.

# 5

# Analysis

The first aspect to note when analysing the work I have produced is that geometric algebra is far more complex than linear algebra in terms of understanding its underlying concepts. I believe there are two main reasons for this. First of all, we are taught the basics of linear algebra from a young age and for this reason have a fundamental understanding of it. Trying to comprehend using a 3D space in a different manner becomes quite confusing fairly quickly. The second reason is that multivectors in geometric algebra are not easily visualised. Most notions in linear algebra can be visualised fairly simply as everything is defined in terms of vectors and scalars. However, the geometric interpretation of a multivector is not so straight forward.

Geometric algebra is not a particularly widely know subject. You would have to assume that the general user would tend to think with a linear approach. Therefore, if it were to be implemented fully within computer graphics, I believe its workings would have to be hidden behind what appears to be linear algebra for it to be truly accessible.

The ray tracer I have produced is extremely primitive. This was mainly due to the fact that most of my time was spent on both the research stage, and implementing the geometric algebra library. However, it became clear to me that the geometric product is a very powerful tool. Virtually all of the operations used by the ray tracer are in some form the geometric product.

Although this did prove to be very useful, there was nothing in the ray tracer that could not have been done in linear algebra. More so to the point, there did not appear to be any aspect of the ray tracer that is performed with less computational power than if we were to use linear algebra.

This suggests that the use of 3-dimensional geometric algebra has no advantage over its linear counterpart.

One of the most interesting aspects of geometric algebra is that the same principles apply regardless of how many dimensions are defined. For example, if we were to move into a 4-dimensional space, a multivector would consist of a scalar, 4 vectors, 6 bivectors, 4 trivectors, and a 4-blade. In order to find the geometric product of two multivectors in this space, we would simply have to extend our definition of the product to account for the new basis blades.

This concept is in fact out of the scope of my assignment. I specified that I would analyse the use of geometric algebra in 3-dimensions. However, during my research it became apparent that this is where the real power of geometric algebra lies. It has been proven that we are able to define 3-dimensional primitives using 5-dimensional geometric algebra. From what I understand of this concept, it appears that calculating things such as intersections becomes extremely trivial. Although, it appears that 3-dimensional geometric algebra has no real benefits over linear algebra, it does seem that if we were to use a space of higher dimensions, we would find numerous advantages.

# 6

# Conclusion

In conclusion, I believe that geometric algebra in 3-dimensions has no significant benefits or disadvantages over linear algebra. It simply provides a different technique with which to implement tools such as a ray tracer.

However, I do believe that with further investigation into the geometric algebra of higher dimensions, computational times could be reduced. This statement is based on papers I have read when performing my research. These can be found in the reference section of this document.

# 7

# Evaluation

I feel this project has been a success in terms its outcome. I set out to gain an understanding of geometric algebra and to find it relevance in computer graphics through the implementation of a ray tracer. I believe I have achieved this to the best of my ability within the time given.

I knew from the outset that to learn the fundamentals of geometric algebra would be a heavy task, let alone implementing it within a C++ library and applying its principles to a ray tracer. Indeed, there were points during this assignment when I felt the latter two tasks would not be achievable. I believe that I could have produced an entire assignment based around the research of this branch of mathematics. For this reason, even though the ray tracer is extremely primitive, I am proud of the amount of work that I have accomplished in the time frame and believe that I have achieved what I set out to.

There are numerous aspects about the project that I would alter if I were to redo the assignment. The code for my program is not particularly efficient or as well presented as I would like. However, my main priority was to produce a geometric algebra library and a program that worked, and I have achieved this.

Early in the project, I think I was over ambitious about the type of ray tracer that I would be able to implement. When I was writing the geometric algebra library, I managed to achieve rotations and reflections about an arbitrary point thinking that I may be able to introduce a lighting model. This took some time that, on reflection, would have been better spent implementing a more efficient ray tracer, or a more proficient library.

Oleg and I worked closely together in the research stage of our projects and aided each other in the implementations of our libraries. I feel that we would have both found it highly beneficial to have worked together on the entire assignment. We could have produced a single library in a shorter space of time and then collectively produced a ray tracer that would have been far more advanced.

There are two main ways in which I could like to extend this project. Firstly, I could make major improvements on the ray tracer that I have designed. There is very little optimisation used and so computational times are extremely large. I also have all the aspects in place with which to create a lighting model for a smooth shaded object. However, this may prove to rather unproductive since we have already established that 3-dimensional geometric algebra has no advantages over linear algebra.

A more appropriate extension would be to further my investigation into the geometric algebra of higher dimensions. I have stated in my conclusion that I believe that in a higher dimension, geometric algebra could prove to have faster computational times than linear algebra. To test this would be the logical extension to this assignement.

# 8

# References

[1] Jaap Suter, “Geometric Algebra Primer”, March 2003
www.jaapsuter.com

[2] Leo Dorst, Stephen Mann, Tim Bourna, “GABLE : A Metalab Tutorial for Geometric Algebra“, December 2002
http://www.cgl.uwaterloo.ca/~smann/GABLE/

[3] Jaap Suter, “Geometric Algebra Primer”, March 2003
www.jaapsuter.com

[4] Jaap Suter, “Geometric Algebra Primer”, March 2003
www.jaapsuter.com
The proofs in this section were lifted straight from this paper.

[5] Leo Dorst, Stephen Mann, Tim Bourna, “GABLE : A Metalab Tutorial for Geometric Algebra“, December 2002
http://www.cgl.uwaterloo.ca/~smann/GABLE/
The equation here is lifted straight from this work

[6] www.imagemagick.org/script/index.php

Other papers, websites, and programs I have found particularly useful:

GAViewer, “Geometric Algebra : The Framework for Geometric Computations”, 2003
http://www.science.uva.nl/ga/tutorials/GDC2003.html
Much of my research was based around this source.

Carlos Rodríguez
http://omega.albany.edu:8008/mat220dir/ga3d-dir/GA3d.html

Martin John Baker
http://www.euclideanspace.com/maths/algebra/clifford/d3/index.htm

# Appendix 1

# Ray Tracer Algorithm

In this appendix I will describe the ray tracer that I have implemented.

The ray tracer is a simple one that will essentially render out the alpha channel of an object.
The camera position is predefined as being at the origin and its orientation is along the positive z-axis.

The main principle behind my ray tracer is that a single ray is produced for each pixel in an image. This is fired from the viewing point through the centre of the pixel. If, the ray intersects with an object, that pixel is coloured white.

The following is the very basic pseudo-code for the ray tracer I have implemented:

```
For each ray
{
        For each polygon
        {
                Find the plane on which the polygon lies
                If the ray intersects with the plane
                {
                        If the point of intersection is within the polygon
                        {
                                Colour the pixel
                                Break out of for each polygon loop
                        }
                }
        }
}
```

# Appendix 2

# Using the Ray Tracer

The source code and executable for my program is found in the folder entitled ray_tracer. In this folder are four further folders:

- classes - This contains header files for the classes I have created. The _class.cpp files contain the definitions of the methods for the corresponding class.

- headers – This folder holds the header files for the definition of all the functions used within the program. The main geometric algebra implementation can be found in operators.h, functions.h, and transformations.h.

- obj – The obj folder contains some .obj files that can be tested with the ray tracer. Further .obj's can be added, although I must stress that once the polygon count get too large (over around 120) the program is slow and temperamental. Any further .obj files that are used must be stored in this folder for the ray tracer to work. I would suggest also that the objects are translated slightly along the positive z-axis in order for the predefined camera to see them.

- images – This is the destination of any rendered images.

The ray tracer can be run from the command line with the following command and arguments:

./raytrace *file_name.obj image_name.jpg x_resolution y_resolution*

An example of this is given below:

./raytrace star.obj star_alpha.jpg 400 400

This will use the object from star.obj to create and image called star_alpha.jpg that has a resolution of 400x400 pixels. star_alpha.jpg will be found in the folder images once the ray tracer is complete.

Please note, that if the resolution gets too high, the ray tracer becomes very laboured. This is especially so with a mesh that contains over around 120 polygons. It is possible to get out larger images but it is best to stick below a resolution of 1000x1000. I would recommend 300x300 for simple tests.

Once the ray tracer has been set going, the user can keep track of its progress. The program will print ascending numbers to the screen. When this reaches the value that the user has set for the resolution in the y axis, the render should be complete. At this point, an openGL window will open to show the position of the object in 3d space relative to the camera and the rays that have intersected with the object. The user can then use the mouse to traverse about the scene.