

Gaigen: a Geometric Algebra Implementation Generator

Daniël Fontijne, Tim Bouma, Leo Dorst
University of Amsterdam

July 28, 2002

Abstract

This paper describes an approach to implementing geometric algebra. The goal of the implementation was to create an efficient, general implementation of geometric algebras of relatively low dimension, based on an orthogonal basis of any signature, for use in applications like computer graphics, computer vision, physics and robotics.

The approach taken is to let the user specify the properties of the geometric algebra required, and to automatically generate source code accordingly. The resulting source code consist of three layers, of which the lower two are automatically generated. The top layer hides the implementation and optimization details from the user and provides a dimension independent, object oriented interface to using geometric algebra in software, while the lower layers implement the algebra efficiently. Coordinates of multivectors are stored in a compressed form, which does not store coordinates of grade parts that are known to be equal to 0. Optimized implementations of products can be automatically generated according to a profile analysis of the user application.

We present benchmarks that compare the performance of this approach to other GA implementations available to us and demonstrate the impact of various settings our code generator offers.

1 Introduction

Geometric algebra promises to provide a powerful computational framework for geometric computations. It contains all geometric operators, primitives of any dimension (not just vectors), and permits specification of geometric constructions in a totally coordinate free matter. We assume the reader has a moderate knowledge of geometric algebra. Some introductions to the subject are [2], [3], [5] and [14]. The reader may benefit from reading *the making of GABLE* [1] before reading this paper, since Gaigen builds on the foundations laid by GABLE. Some programming knowledge is also required to fully understand some implementation and performance details.

To show that geometric algebra is not only a good way to think and reason about geometric problems as they appear in computer science, but also a viable way to implement solutions to these problems on computers, an implementation that is efficient with respect to both computational and memory

resources is required. It may be true that using geometric algebra helps you to solve and implement your problems faster and more easily, while making fewer mistakes, but if this advantage comes at a significant speed or memory penalty, people working on hard real-world applications will quickly, perhaps rightly, turn away and stick with their current solutions.

An example of inefficient use of memory and computational resources occurs when one treats 5 dimensional multivectors (such as used in the conformal model of 3D geometry [4]) in a naive way. The representation of a 5 dimensional multivector requires $2^5 = 32$ coordinates. When two of these multivectors are multiplied naively (using the approach in [1]), $O(d^2)$ floating point operations (i.e. 32^2) are required. In most applications however, many of these coordinates will be 0 (typically at least half of them). By not storing these coordinates, memory usage is reduced. Moreover, fewer floating point operations are required if it is known which coordinates are 0, since we don't have to multiply and sum terms that we know will be 0 anyway. Our approach exploits this insight by not storing the coordinates of grade parts that are known to be 0, thus reducing memory usage and increasing processing efficiency.

We also wanted Gaigen to be portable between platforms, without losing performance. Naturally, the C++ code generated by Gaigen is fully portable to all platforms for which a C++ compiler is available, but many mainstream platforms support SIMD (Single Instruction Multiple Data) floating point instructions (such as SSE [8], 3DNow! [7] and AltiVec [9]) aimed at doing floating point computations faster. To allow individual users to optimize Gaigen for their particular platform, the lowest layer of source code generated by Gaigen, that performs basic operations such as multiplication, addition and copying, was designed to be easily replaced by a version optimized for a particular platform.

At the same time we desired a sufficiently general package for computing with geometric algebra and a successor to GABLE [1][2]. However optimized or efficient the new package may be, it should be straightforward to use, support any dimension and basis vector signature and be extensible.

This paper describes how we combined these desires into our geometric algebra package called Gaigen. Gaigen *generates* C++ source code for a specific geometric algebra, according to the user's needs. Here are some properties of the algebra that the user can specify:

- dimension,
- signature of the orthogonal basis vectors (+1, 0, -1),
- reciprocal null vectors
- what products to implement (geometric product, (modified) Hestenes inner product, left and right contraction, outer product, scalar product)
- how to optimize the implementation of these products,
- the storage order of the coordinates,
- what extra functions to implement (e.g. reverse, addition, inverse, projection, rejection, outermorphism) and
- what coordinate memory (re-)allocation method should be used.

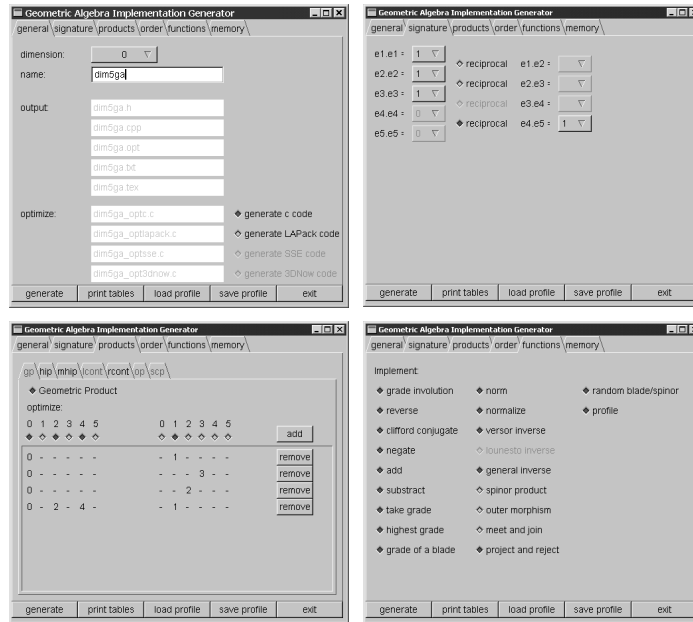


Figure 1: Screenshots of the Gaigen user interface, showing how the user can select name, dimension, basis vector signature, products and functions of the desired geometric algebra.

Usage of Gaigen is described in a separate user manual [6]. Some screenshots of Gaigen’s user interface are shown in figure 1.

The rest of this paper explains how Gaigen represents and stores multivectors (section 2), how multiplication tables for products of multivectors from arbitrary geometric algebras are constructed using the binary representation of basis elements (section 3) and how the optimized products are implemented (section 4). These are all problems that Gaigen handles in a specific, performance minded way. Furthermore the paper describes algorithms that Gaigen uses to efficiently compute inversions, outer morphisms, duals, factorizations of blades and versors and the meet and join. These subjects might be of interest to anybody implementing a GA package. The paper ends with a benchmarks of the performance of Gaigen (section 11) and discussion of Gaigen in general (section 12).

A short note on notation: we will use **bold font** for multivector quantities, and *math* font for scalars. The variable d will be the dimension of the geometric algebra at hand throughout the whole paper. We will use A, B, C and D as general multivector variables and E, F, G and H as basis element variables. When we say that grade g of a multivector is empty, we mean that the grade g part of the multivector is equal to 0.

2 Representation of multivectors

Gaigen represents a multivector as an array of coordinates relative to an orthogonal basis of the required geometric algebra. Suppose we want to implement a 2D geometric algebra on the basis $[1, e_1, e_2, e_1 \wedge e_2]$, then a multivector \mathbf{A} could be represent as an array of four coordinates $[A_s, A_1, A_2, A_{12}]$, that relate to the multivector \mathbf{A} as follows:

$$\mathbf{A} = [1, e_1, e_2, e_1 \wedge e_2] \begin{bmatrix} A_s \\ A_1 \\ A_2 \\ A_{12} \end{bmatrix}, \quad (1)$$

If we define $[\mathbf{A}] = [A_s, A_1, A_2, A_{12}]^T$, then we can we denote this representation in shorthand as:

$$\mathbf{A} \hat{=} [\mathbf{A}] \quad (2)$$

2.1 Coordinate Order and Basis Element Orientation

Gaigen allows the user to specify in which order the coordinates of the column matrix on the right hand side of equation 1 are stored. The only restriction is that the coordinates are ordered by grade: coordinates for grade n always precede coordinates for grade $n + 1$ in the coordinate array. This restriction simplifies the compression of coordinates as described in section 2.2.

The coordinates of one specific grade however, can be stored in any order. The reason for this is the lack of a natural order for the coordinates beyond dimension 3. This can be shown with a short example. We could store the coordinates of a 3D geometric algebra as follows:

$$\begin{array}{c} A_s, \\ A_1, \quad A_2, \quad A_3, \\ A_{23}, \quad A_{31}, \quad A_{12}, \\ A_{123}, \end{array}$$

with the motivation that all coordinates are ordered according to grade, that the vectors coordinates are ordered according to their index, that for each vector coordinate it's dual bivector coordinate is beneath it and the bivectors are in a cyclic order. However, we have not found such strong motivations for geometric algebras of higher dimension.

We can conclude there is no reason for *us* to enforce a specific order. The *user* however, may have a preference for the order of the coordinates. The source of the coordinates (e.g., real world measurements) may have a specific order. When the user enters these coordinates into multivector variables, it would be efficient if the user does not have to reorder them. Thus Gaigen allows for arbitrary ordering of the basis elements.

The user can also specify the *orientation* of the basis elements, e.g., decide whether $e_1 \wedge e_2$ or $e_2 \wedge e_1$ is the orientation to which the bivector coordinate A_{12} (or A_{21}) refers. Again, this option is provided because there is no reason for Gaigen to enforce a specific orientation of basis elements upon the user, and some users may have their data in a specific format.

2.2 Compressed coordinate storage

Gaigen does not always store all coordinates of a multivector. For every multivector that is created, Gaigen keeps track of the grade parts that are empty. Every multivector coordinate array is accompanied by a bit field that specifies the *grade part usage*. When bit n of the bit field is on, this indicates that the grade part i coordinates are stored in the coordinate array of A ; when bit i is off, it indicates that the grade part i coordinates are omitted from the array.

This storage scheme can significantly reduce the amount of memory required to store a multivector. When only even or odd multivectors (e.g., versors) are used, already only half the amount of memory is required. For blade storage, the scheme is even more efficient.

2.3 Tracking compressed coordinate storage

Assume that when a multivector is created, Gaigen always knows its grade part usage. Now we would like to track that grade part usage through any function f (e.g. a geometric product, or a dualization) that we want to perform on multivectors: when some function f is applied to one or two multivectors, like:

$$B = f(C) \tag{3}$$

or

$$B = f(C, D) \tag{4}$$

we want to know (preferably beforehand) what the grade part usage of the result B will be. So, whenever possible, Gaigen first computes the expected grade part usage of B . Gaigen then allocates enough memory to store the coordinates of those grades, computes the result, stores it in the compressed coordinate array, and sets the grade usage bit field of B .

It may not always be possible for Gaigen to compute efficiently what the grade part usage of B will be. Then Gaigen assumes all grades will be non-zero, computes the result and then *compresses* the result, leaving out all grades which are empty. Note however that this will be less efficient than knowing beforehand which grade parts in B will be empty.

Functions (especially linear transformations) such as $x' = \tilde{R}xR$ where x is a vector and R a rotor, present a significant problem to this storage method. The product $\tilde{R}x$ is an odd multivector: it may have odd grade parts larger than 1 that are not equal to 0. The product $(\tilde{R}x)R$ however, is supposed to be of grade 1, because the grade parts higher than 1 cancel out when $\tilde{R}x$ is multiplied by R . Due to floating point round off errors however, this is not necessarily true in Gaigen. The product $(\tilde{R}x)R$ may have odd grade parts larger than 1 that are not empty, though they are very small. Gaigen lacks the ability to predict that a product such as $\tilde{R}xR$ will be of grade 1.

We have implemented two solutions to this problem. First of all, Gaigen has an outermorphism operator built into it (section 6). Any linear transformation L can be stored in a matrix form \underline{L} . The product $\underline{L}A$ is guaranteed to be grade preserving, so all grades that are empty in A are known to be empty in $\underline{L}A$. Secondly Gaigen supports a grade selection mechanism that can be used to get

rid of unwanted grade parts (i.e., $\langle \tilde{\mathbf{R}}\mathbf{x}\mathbf{R} \rangle_1$, to extract only the vector part from $\tilde{\mathbf{R}}\mathbf{x}\mathbf{R}$).

2.4 Memory (re-)allocation schemes

Because Gaigen tracks what multivector grade parts are empty, it can reduce memory usage. However, there is a trade-off between memory (re-)allocation and computational efficiency. Suppose a C++ program uses a 3D multivector variable A , which is assigned a vector quantity (grade 1, requiring 3 coordinates). When this same variable is later assigned a rotor quantity (grade 0 + grade 2, requiring 4 coordinates), more memory must be allocated to store the extra coordinate. To preserve memory resources, we always want to allocate only the smallest amount of memory required to store the coordinates. However, reallocating memory costs time. Allocating the maximum amount of memory required to store all possible coordinates (2^d , where d is the dimension of the algebra), ensuring that we would never have to reallocate memory, wastes memory resources

To allow the user to make this trade-off between memory usage and computation time, Gaigen currently has four memory allocation schemes. All schemes operate transparently to the user and can be replaced with each other. memory allocation schemes can be changed by selecting another scheme and regenerating the algebra implementation. The allocation schemes are:

Tight: Exactly the right amount of memory is allocated to store the coordinates of the non-zero grades of a multivector. This implies frequent memory reallocation, which is done via a simple and efficient memory heap.

Balanced: To prevent abundant memory reallocation, the balanced allocation scheme does not always free memory that is no longer required for storage, up to a certain waste factor which the user can specify. Suppose a variable holds a 3D rotor (4 coordinates), and is assigned a vector (3 coordinates); the memory waste would be 33.3% if 4 coordinates memory locations would be used to store 3 coordinates. If the waste factor were larger or equal to $\frac{4}{3}$, the balanced allocation scheme would decide not to reallocate the memory.

Maximum: The maximum number of memory locations to store all 2^d coordinates is allocated when a multivector variable is created. Gaigen never has to reallocate memory.

Maximum parity pure: We call a multivector parity pure if it is either odd or even. If the dimension of the algebra is larger than 0, only half of the 2^d coordinates have to be allocated to store the coordinates of a parity pure multivector variable. The user must guarantee that he will never create multivectors which are not parity pure, or weird things can happen (a crash or incorrect results). Non-parity pure multivectors never arise if all multivectors are constructed as products of blades.

product	condition(s)
outer product	$\text{grade}(\mathbf{G}) = \text{grade}(\mathbf{E}) + \text{grade}(\mathbf{F})$
scalar product	$\text{grade}(\mathbf{G}) = 0$
left contraction	$\text{grade}(\mathbf{G}) = \text{grade}(\mathbf{F}) - \text{grade}(\mathbf{E})$
right contraction	$\text{grade}(\mathbf{G}) = \text{grade}(\mathbf{E}) - \text{grade}(\mathbf{F})$
Hestenes inner product	$\text{grade}(\mathbf{G}) = \text{grade}(\mathbf{F}) - \text{grade}(\mathbf{E}) $
modified Hestenes inner product	$\text{grade}(\mathbf{G}) = \text{grade}(\mathbf{F}) - \text{grade}(\mathbf{E}) ,$ $\text{grade}(\mathbf{E}) \neq 0, \text{grade}(\mathbf{F}) \neq 0$

Figure 2: Conditions for deriving products from the geometric product. If the result of a geometric product $\mathbf{G} = \mathbf{E}\mathbf{F}$ (both \mathbf{E} and \mathbf{F} must be *basis elements*) agrees with all conditions for one of the products p listed in this table, then the product p of \mathbf{E} and \mathbf{F} is equal to the geometric product of \mathbf{E} and \mathbf{F} . Otherwise, it is 0. The conditions in the table are only valid for multiplying orthogonal basis elements.

3 Implementing the GA products

After the coordinate storage problem has been solved efficiently, the next logical step in implementing a geometric algebra is probably computing the various products. Gaigen currently supports 7 products: the geometric product, the outer product, the scalar product and four variations of the inner product.

3.1 Deriving products from the geometric product

The geometric product is the fundamental product of geometric algebra. All other products can be derived from it. This is not only true in theory, but is also applied in practice by Gaigen. As explained below, during the source code generation phase, Gaigen only needs to be able to compute the geometric products of arbitrary basis elements e_α . The conditions in figure 2 are used to derive the other products from the geometric product.

3.2 Computing the geometric product

A geometric product of two multivectors \mathbf{A} and \mathbf{B} is essentially a linear transformation of $[\mathbf{B}]$ by some matrix $[\underline{\mathbf{A}}^G]$. The matrix $[\underline{\mathbf{A}}^G]$ linearly depends on \mathbf{A} . Once we have the appropriate matrix $[\underline{\mathbf{A}}^G]$, we can state that

$$\mathbf{AB} = [\underline{\mathbf{A}}^G][\mathbf{B}] \quad (5)$$

Thus we see that a d dimensional geometric algebra is actually a 2^d dimensional linear algebra. However, in geometric algebra we assign geometric meaning to the elements of the algebra, while a general 2^d dimensional linear algebra does not have such an interpretation. The geometric interpretation of the elements of the algebra also suggests that there is a lot more structure in a d dimensional geometric algebra than in a 2^d dimensional linear algebra, which Gaigen tries to exploit for efficiency.

Gaigen is able to compute matrices $[\underline{\mathbf{A}}^G]$ automatically. During the extension of 3D GABLE [1] to 4D and 5D it proved tedious and error-prone to con-

struct these $2^d \times 2^d$ matrices by hand. Here is an example of a matrix expansion of a multivector \mathbf{A} into a matrix $[\underline{\mathbf{A}}^G]$ for a 3D geometric algebra with a Euclidean signature:

$$[\underline{\mathbf{A}}^G] = \begin{bmatrix} +A_s & +A_1 & +A_2 & +A_3 & -A_{12} & -A_{13} & -A_{23} & -A_{123} \\ +A_1 & +A_s & +A_{12} & +A_{13} & -A_2 & -A_3 & -A_{123} & -A_{23} \\ +A_2 & -A_{12} & +A_s & +A_{23} & +A_1 & +A_{123} & -A_3 & +A_{13} \\ +A_3 & -A_{13} & -A_{23} & +A_s & -A_{123} & +A_1 & +A_2 & -A_{12} \\ +A_{12} & -A_2 & +A_1 & +A_{123} & +A_s & +A_{23} & -A_{13} & +A_3 \\ +A_{13} & -A_3 & -A_{123} & +A_1 & -A_{23} & +A_s & +A_{12} & -A_2 \\ +A_{23} & +A_{123} & -A_3 & +A_2 & +A_{13} & -A_{12} & +A_s & +A_1 \\ +A_{123} & +A_{23} & -A_{13} & +A_{12} & +A_3 & -A_2 & +A_1 & +A_s \end{bmatrix}$$

3.3 Computing matrix expansions

To compute matrix expansions $[\underline{\mathbf{A}}^G]$ we use the following rule. If the geometric product of two orthogonal basis elements \mathbf{e}_α and \mathbf{e}_β

$$\mathbf{e}_\alpha \mathbf{e}_\beta = c\mathbf{e}_\gamma \quad (6)$$

(where $c \in \{-1, 0, 1\}$), then

$$[\underline{\mathbf{A}}^G]_{\gamma,\beta} = cA_\alpha. \quad (7)$$

For if we had stored the basis elements \mathbf{e}_α , \mathbf{e}_β , \mathbf{e}_γ as general multivectors $[\mathbf{A}]$, $[\mathbf{B}]$, $[\mathbf{C}]$, then the coordinate A_α , that refers to \mathbf{e}_α must have been in column β and row γ of $[\underline{\mathbf{A}}^G]$: it must be in column β such that A_α gets multiplied with coordinate B_β ; it must be in row γ such that the result get added to coordinate C_γ . This is illustrated in figure 3.

The signature of the basis vectors and the orientation of the basis elements are responsible for the sign c , as discussed below in section 3.4.5.

By computing the geometric product of every combination of basis elements, Gaigen is able to build up a symbolic matrix expansion for the geometric product, which is then used during code generation.

3.4 Computing the geometric product of basis elements

The last step that remains in automatically computation symbolic matrix expansion for the products is to compute the geometric product of arbitrary orthogonal basis elements.

3.4.1 The binary representation of basis elements

We show how to compute the product of any two basis elements making use of what we call the binary representation of basis elements. The binary representation is a mapping between basis elements such as \mathbf{e}_1 , $\mathbf{e}_3 \wedge \mathbf{e}_4$ and $\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3$ and binary numbers $n_b \in \mathbb{N}$. Every digit (or bit) in a binary number represents the presence or absence of a basis vector in a basis element. We use the least significant bit 2^0 for \mathbf{e}_1 , the next bit 2^1 for \mathbf{e}_2 and bit 2^{d-1} for \mathbf{e}_d . A separate scalar value $\alpha \in \{-1, 0, +1\}$ represents the sign of the basis element (we use

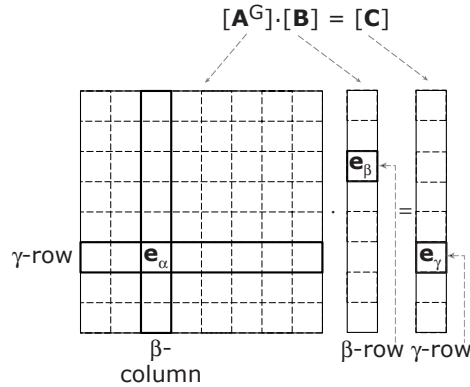


Figure 3: How to build a matrix expansion for the geometric product given $e_\alpha e_\beta = ce_\gamma$.

lowercase greek letters for these scalar variables). Here are some examples of basis elements and their binary representations:

$$e_1 \rightleftharpoons +1 * 1_b \quad (8)$$

$$e_1 \wedge e_2 \wedge e_3 \rightleftharpoons +1 * 111_b \quad (9)$$

$$e_3 \wedge e_4 \rightleftharpoons +1 * 1100_b \quad (10)$$

$$e_4 \wedge e_3 \rightleftharpoons -1 * 1100_b \quad (11)$$

$$1 \rightleftharpoons +1 * 0_b \quad (12)$$

$$0 \rightleftharpoons 0 * x_b, x_b \in \mathbb{N} \quad (13)$$

We use a subscript 'b' to indicate binary numbers. The expression $\eta * E_b$ is shorthand for the binary representation of a basis element E ; this notation implicitly defines the multivector E to be a basis element, since the binary representation only works for basis elements. Notice equations 12 and 13. In the binary representation we use the binary number 0_b to represent the unit scalar valued basis element 1. The value 0 can be represented by any binary number, as long as it is combined with a zero sign. Also notice equations 10 and 11. Although the binary numbers representing the basis elements are identical (because both $e_3 \wedge e_4$ and $e_4 \wedge e_3$ contain the same basis vectors), they have a different sign, because their orientation is opposite.

As can be seen in figure 4 the binary representation scales nicely with the dimension of the algebra; each time the dimension of the algebra increases, an extra bit is required to represent the new basis vector. The basis elements of a 1D algebra can be represented by a 1 digit binary number. A 2D algebra requires 2 bits, a d dimensional algebra requires d bits.

basis element	bin. rep.
$\mathbf{1}$	0_b
\mathbf{e}_1	1_b
\mathbf{e}_2	10_b
$\mathbf{e}_1 \wedge \mathbf{e}_2$	11_b
\mathbf{e}_3	100_b
$\mathbf{e}_1 \wedge \mathbf{e}_3$	101_b
$\mathbf{e}_2 \wedge \mathbf{e}_3$	110_b
$\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3$	111_b
\mathbf{e}_4	1000_b

Figure 4: The scaling of the binary representation with the dimension of the algebra.

3.4.2 Computing the geometric product of basis vectors using the binary representation

The geometric product of two basis elements $\mathbf{H} = \mathbf{F} \mathbf{G}$ is computed as a binary *exclusive or* (\otimes) of the binary representations (ϕ, \mathbf{F}_b) and (γ, \mathbf{G}_b) :

$$\mathbf{H} = \mathbf{F} \mathbf{G} \Leftrightarrow (\chi, \mathbf{H}_b) = (\phi * \gamma * \sigma * \rho, \mathbf{F}_b \otimes \mathbf{G}_b) \quad (14)$$

Here ϕ , γ and $\chi = \phi * \gamma * \sigma * \rho$ are the signs of \mathbf{F} , \mathbf{G} and \mathbf{H} respectively. The hard part is computing the values of σ and ρ . The value of σ is equal to the product of the signatures of all basis vectors that were annihilated during the product. ρ accounts for the *reordering* of basis vectors that is required before any basis vectors can be annihilated. How to compute σ and ρ is explained below. But first we show why we can compute which basis vectors are present in the result of a geometric product of basis elements using a simple exclusive or.

We start by noting that \mathbf{F} and \mathbf{G} can be written as

$$\begin{aligned} \mathbf{F} &= \mathbf{e}_{f_1} \mathbf{e}_{f_2} \cdots \mathbf{e}_{f_n} \\ \mathbf{G} &= \mathbf{e}_{g_1} \mathbf{e}_{g_2} \cdots \mathbf{e}_{g_m} \end{aligned}$$

where the \mathbf{e}_i are all orthogonal basis vectors (the special case of reciprocal null vectors is treated in section 3.4.5). Due to orthogonality of the basis vectors, the geometric product for each \mathbf{e}_i can be computed independently of the others. For each basis vector \mathbf{e}_i , there are four possible cases:

- \mathbf{e}_i is present in both \mathbf{F} and \mathbf{G} . In this case the basis vector gets annihilated (it will not be present in the result) and only contributes its signature $\sigma_i = \mathbf{e}_i^2$ to the sign of the result. The respective bit 2^{i-1} is 1 in both \mathbf{F}_b and \mathbf{G}_b , and since $1 \otimes 1 = 0$, this is handled correctly by the exclusive or operation.
- \mathbf{e}_i is present in either \mathbf{F} or \mathbf{G} , so \mathbf{e}_i will be present in the result. Bit 2^{i-1} is 1 in either \mathbf{F}_b or \mathbf{G}_b and 0 in the other, and since $1 \otimes 0 = 0 \otimes 1 = 1$, this case is also handled correctly by the exclusive or operation.
- \mathbf{e}_i is present in neither \mathbf{F} nor \mathbf{G} . Then \mathbf{e}_i will also not be present in the result. Bit 2^{i-1} is 0 in both \mathbf{F}_b and \mathbf{G}_b , and since $0 \otimes 0 = 0$, this case is again handled correctly by the exclusive or operation.

3.4.3 Computing the value of σ

As said above, σ is equal to the product of the signatures of all basis vectors that were annihilated during the product. Thus computing the value of σ can be done as follows:

$$\sigma = \prod \sigma_i, \text{ for all basis vectors } e_i \text{ annihilated in the product.} \quad (15)$$

We note that $\sigma \in \{1, 0, -1\}$, since the signatures σ_i of the basis vectors are also element of $\{1, 0, -1\}$. If no basis vectors are annihilated, then we set $\sigma = 1$.

3.4.4 The basis vector reordering sign ρ

When we compute the geometric product of two basis elements \mathbf{F} and \mathbf{G} , the basis vectors from \mathbf{F} and \mathbf{G} have to into a specific order such that identical basis vectors from \mathbf{F} and \mathbf{G} are next to each other. For example, suppose that $\mathbf{F} = e_1 e_2 e_3$ and $\mathbf{G} = e_1 e_2$. Then

$$\mathbf{F} \mathbf{G} = e_1 e_2 e_3 e_1 e_2 \quad (16)$$

$$= -e_1 e_1 e_2 e_2 e_3 \quad (17)$$

The basis vectors are in the specific order we want in equation 17. Note the sign has flipped in equation 17. The sign changes every time we 'flip' the order of two neighboring basis vectors. In this example, an odd number of flips is required, so the sign is negative. This sign change is exactly the sign ρ that we still have to compute in order to complete the computation of the geometric product of basis elements.

Given the binary representations $\mathbf{F}_b, \mathbf{G}_b$ of arbitrary basis elements \mathbf{F}, \mathbf{G} , ρ , can be computed using the following equations:

$$c(\mathbf{F}_b, \mathbf{G}_b) = \sum_{i=0}^{d-1} \left(\frac{\mathbf{F}_b \& 2^i}{2^i} \left[\sum_{j=0}^{i-1} \frac{\mathbf{G}_b \& 2^j}{2^j} \right] \right) \quad (18)$$

$$\rho(\mathbf{F}_b, \mathbf{G}_b) = -1^{c(\mathbf{F}_b, \mathbf{G}_b)} \quad (19)$$

where $\&$ is the binary *and* operator.

The motivation for the function c (count) is as follows. We want to count how many basis vector flips are required to get the basis vectors of \mathbf{F} and \mathbf{G} into the correct order. For every bit i that is 1 in \mathbf{F}_b we want to count how many bits $j, j < i$ are 1 in \mathbf{G}_b . We count how many bits $j, j < i$ are 1 in \mathbf{G}_b with this subexpression: $\left[\sum_{j=0}^{i-1} \frac{\mathbf{G}_b \& 2^j}{2^j} \right]$. Then we use the subexpression $\frac{\mathbf{F}_b \& 2^i}{2^i}$ to either in- or exclude the count depending on whether the bit i is 1 in \mathbf{F}_b .

If the number of flips is even, the $\rho = 1$, otherwise $\rho = -1$. In practice, the function c is more easily implemented using binary 'bit shift' and binary 'and' operations, than by directly implementing equation 18.

3.4.5 Reciprocal Null Vectors

Besides basis vectors of any signature, Gaigen also supports pairs of reciprocal null vectors, which are two null vectors which act as the inverse of the other.

We now show how to extend the binary representation and the rules for computing the products of basis elements given above to incorporate reciprocal null vectors.

In Gaigen, a pair of reciprocal null vectors is a pair of basis vectors e_i and e_j for which the following is true:

$$e_i^2 = e_j^2 = \mathbf{0} \quad (20)$$

$$e_i e_j = \sigma_{ij} + e_i \wedge e_j \quad (21)$$

$$j = i + 1 \quad (22)$$

where $\sigma_{ij} \in \{-1, 1\}$. The requirement that e_i and e_j are 'neighbours' (equation 22) will simplify matters below. The following table is a multiplication table for the geometric product in a 2D geometric algebra with reciprocal null vectors e_1 and e_2 :

	1	e_1	e_2	e_{12}
1	+1	+1	$+e_2$	$+e_{12}$
e_1	$+e_1$	0	$\sigma_{12} + e_{12}$	$-e_1$
e_2	$+e_2$	$\sigma_{12} - e_{12}$	0	$+e_2$
e_{12}	$+e_{12}$	$+e_1$	$-e_2$	+1

We mentioned above that we can treat each basis vector independently of the others while computing a geometric product of basis elements because all basis vectors are orthogonal. Of course, this isn't true anymore when pairs of reciprocal null vectors must be implemented, since they are not orthogonal. However, pairs of reciprocal null vectors are always orthogonal to all other basis vectors. So we can handle all each pair of reciprocal null vectors and each ordinary basis vector independently of all others.

To handle reciprocal null vectors, we have to adapt the three sub problems of computing the geometric product of basis elements as treated above: computing which basis vectors will be present in the result, computing the contribution sign σ and computing the reordering sign ρ .

To compute which basis vectors are present in the result of a geometric product of basis elements, we use a lookup table very much like the multiplication table above. The lookup table states for each possible input what basis vectors will be present in the result, and what the sign is contributed to σ . Note that in some cases, the lookup table can contain a sum of two the basis elements. This is due to the fact that the geometric product of two reciprocal null vectors can result in the sum of a scalar and a bivector. E.g. in the table above $e_1 e_2 = \sigma_{12} + e_{12}$. Thus in general, the geometric product of two basis elements results in a sum of basis elements. Gaigen implements this by storing a separate binary representation of each term in the sum. The maximum number of basis elements in this sum is 2^n , where n is the number of pairs of reciprocal null vectors in the algebra.

Computation of the reordering sign ρ changes slightly due to the introduction of reciprocal null vectors. Reciprocal null vectors always have to remain neighbours, hence they always are reordered together. Flipping the order of an ordinary basis vector and a pair of reciprocal null vectors can be simplified to flipping the order of two ordinary basis vectors. A pair of reciprocal null vectors is then represented by a *single* bit in the binary representation. This bit

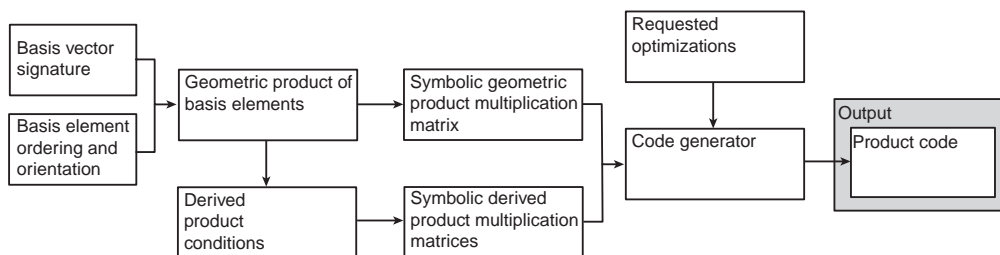


Figure 5: Summary of implementing the GA products.

is 0 if an even number of reciprocal null vectors from the pair is present in the basis elements, and the bit is 1 if an odd number of reciprocal null vectors is present. The binary representation can then be fed to equations 18 and 19 to give the correct reordering sign ρ .

3.5 Summary of implementing the GA products

Figure 5 summarizes the process in which Gaigen generates code that implements the geometric algebra products. The input to the pipeline is the basis vector signature and the ordering and orientation of the basis elements (section 2). Using this information, we are able to compute the geometric product of basis elements as described in section 3.4. By applying the rules from figure 2, we are also able to derive the other products of basis elements. We use equation 7 to fill up the symbolic product multiplication matrices. The code generator takes these symbolic matrices and the requested optimizations (to be discussed in section 4) and outputs the C++ source code that implements the products.

4 Optimized product implementation

When Gaigen computes the product $C = p(A, B)$, where p can be any product Gaigen supports, it only multiplies and sums terms which are known not to be equal to 0. The straightforward way to implement this is for each group of terms which involve grade i coordinates from $[A]$ and grade j coordinates from $[B]$, to check whether either of these grade parts is 0. However, a general product involves $O(d^2)$ combinations of grades from A and B , and thus $O(d^2)$ checks are required. This results in a lot of conditional jumps, which causes a performance hit for modern pipelined processors.

Gaigen works around this problem by allowing the user to specify a set of product/multivector combinations for which optimized implementations should be generated. For instance, the user can tell Gaigen to optimize the geometric product of a vector and a rotor. Gaigen will then generate a piece of code which can only perform the product of that specific combination of multivectors, which does not contain any conditional jumps.

In general it is not feasible to implement an optimized version of every product/multivector combination. This would result in a very large amount

```

Profile information for dim3:
      grade 0123 x 0123
Left Contraction  *.* x *.* : 17.79% (797066 times)
  Outer Product  *.* x *.* : 17.40% (779678 times)
Geometric Product  *.* x *.* : 12.48% (559085 times)
Geometric Product  *.* x *.* : 12.48% (559085 times)
  Outer Product  *.* x *.* : 12.38% (554603 times)
  Outer Product  *.* x *.* : 5.64% (252763 times)
  Scalar Product  *.* x *.* : 5.15% (230618 times)
  Scalar Product  *.* x *.* : 5.06% (226563 times)
Geometric Product  *.* x *.* : 4.49% (201000 times)
  Scalar Product  *.* x *.* : 4.34% (194594 times)

```

Figure 6: Profile of a camera calibration algorithm [11], which uses a 3D geometric algebra. The profile shows which and how often specific combinations of multivectors and products were used in the program. Each line consists of the name of the product, which grades were not empty for each of the operands (below the '0123'), and how often the product/multivector combination was used, both in percentage and absolute. Product/multivector combinations with a usage less than 2% were omitted.

of code, wasting compilation time and disk space and memory. So the user has to select a specific set of product/multivector combinations which are used most often (the theoretical set of 5% of the combinations which is used 95% of the time).

The major problem with this approach is that the user has to specify which product/multivector combinations to optimize. The user would have to search through the program code and find out which product/multivector combinations are used most often. This is practically impossible and best choice might even on the input data of the program. That's why Gaigen can generate a profile of a program at runtime. By letting Gaigen generate code with the 'profile' option turned on, it will include profiling code. The user can then compile and run the application program using a representative set of input data and dump the profiling information at the end of the run. This information will state which product/multivector combinations have been used most often, and can be used to specify which product/multivector combinations to optimize. When optimization of program is finished, the profile option is of course turned off and the Gaigen code regenerated.

Figure 6 shows a profile of a motion capture camera calibration algorithm [11]. From the profile one can read that the left contraction of two vectors is used most often (17.79%), followed by the outer product of a scalar and a vector (17.40%), and the geometric product of a 'grade 1 and 3 spinor' and a rotor (12.48%). By optimizing all product/multivector combinations which are used more than 2% of the time, the computation time of the program decreased by a factor of 2.5 compared to non-optimized products. This profiling information can be used to add optimizations by hand or Gaigen can add optimizations to an algebra specification automatically from a profiling file.

4.1 Dispatching

When a product of two multivectors is computed, Gaigen has to check if there is an optimized function available for that specific combination of product / multivectors. If there is no optimized function available, a general function is called that can compute the product for any pair of multivectors. If there is an optimized function available, that function must be called.

The problem of finding the the right function and executing it is called dispatching. We implemented three dispatching methods in Gaigen:

- *if else*: the grade part usage of one or both of the multivector arguments is compared to the grade part usage required for some optimized product. If there is a match, the function is called, or second test might be required for the grade part usage of the other multivector argument. If no optimized function is found for the arguments, the general function is called. In C++ this is implemented as a `if {} else {}` tree.
- *switch*: a C++ switch is used to jump the appropriate (general or optimized) function. The value on which the switch is made is constructed from the grade part usage of the arguments.
- *lookup table*: The grade part usage of both multivector arguments is transformed into an index into a lookup table. This table contains a pointer to the appropriate (general or optimized) function.

Both the *if else* and *switch* dispatching method can take the relative importance of the product/multivector combinations into account: a product/multivector combination that is used more often will be identified with less instructions than one that is used less often. Benchmarks (see section 11) show that the *if else* method is the most efficient.

5 Inversion

The computation of the inverse A^{-1} of a multivector A is a problem that can be handled in a number of different ways. That's why Gaigen supports a number of different inversion algorithms so the user can pick the algorithm best suited to the application.

First of all Gaigen provides a matrix inversion approach to multivector inversion. Gaigen constructs a $2^d \times 2^d$ matrix $[\underline{A}^G]$ and reduces it to identity using partial pivoting. At the same time it applies all operations it applies to $[\underline{A}^G]$ to a $2^d \times 1$ column vector. If the column vector is initialized to $[1 \ 0 \ 0 \dots 0]^T$, it will end up containing the coordinates of A^{-1} . This inversion method works for *all* invertible multivectors, but is inefficient since it requires $O(2^{d^3})$ floating point operations.

If only versors have to be inverted, a much faster method is possible. Versors are multivectors which can be written as a geometric product of vectors, so $A = v_1 v_2 \dots v_n$. A blade is a homogeneous versor, so this method also works for blades. A nice property of versors is that $A \tilde{A} = s$ where s is scalar and \tilde{A}

is the reverse of \mathbf{A} . Thus if $s \neq 0$

$$\frac{\mathbf{A}\tilde{\mathbf{A}}}{s} = \frac{\mathbf{A}\tilde{\mathbf{A}}}{\mathbf{A}\tilde{\mathbf{A}}} = 1 \implies \mathbf{A}^{-1} = \frac{\tilde{\mathbf{A}}}{\mathbf{A}\tilde{\mathbf{A}}}. \quad (23)$$

This inversion approach, though limited to versors, requires only $O(2^d)$ floating point operations and is thus favorable to the matrix inversion method if only versors are used.

A third inversion method, called the Lounesto inverse, is described in more detail in [10] (pg. 57) and [1]. It only works in three dimensional algebras. It is based on the observation that in three dimensions the geometric product of a multivector \mathbf{A} and its Clifford conjugate $\bar{\mathbf{A}}$ only has two grades, a scalar and a pseudoscalar (the Clifford conjugate is the grade involution of the reverse of a multivector). From this it follows (see the references) that

$$\mathbf{A}^{-1} = \frac{\bar{\mathbf{A}} ((\mathbf{A}\bar{\mathbf{A}})_0 - (\mathbf{A}\bar{\mathbf{A}})_3)}{(\mathbf{A}\bar{\mathbf{A}})_0^2 - (\mathbf{A}\bar{\mathbf{A}})_3^2} \quad (24)$$

A simple test shows that, in Gaigen, using 3D versors, the versor inverse is approximately 1.5 times faster than the lounesto inverse, so the versor inverse should be preferred over the lounesto inverse when only versors are used. In the same benchmark, the general inverse is more than 10 times slower than both the versor and the Lounesto inverse.

6 Outermorphism

Gaigen has an outermorphism operator built in to support efficient linear transformations of multivectors. The outermorphism can be used to

- implement linear transformations which can not easily be represented in term of other geometric algebra operations.
- to ensure linear transformations such as $\mathbf{x}' = \mathbf{R}\mathbf{x}\tilde{\mathbf{R}}$ are grade preserving (the problem that was described in section 2.2).
- to enhance the performance of linear transformations.

An outermorphism is any function f for which the following is true for any pair of input blades \mathbf{a} and \mathbf{b} .

$$f(\mathbf{a} \wedge \mathbf{b}) = f(\mathbf{a}) \wedge f(\mathbf{b}) \quad (25)$$

For this to be true, f must be a linear transformation, and thus it can be represented by a $2^d \times 2^d$ matrix $[\mathbf{F}^O]$. We can then apply the outermorphism to a blade \mathbf{A} like this:

$$[\mathbf{B}] = [\mathbf{F}^O][\mathbf{A}] \quad (26)$$

Since outermorphisms are always grade preserving, we are doing too much work in the above equation. As shown in figure 7 for the 3D case, matrix entries of a limited diagonal band are all 0. Thus Gaigen stores the outermorphism as a set of matrices, one for each grade part. To apply the outermorphism to a blade then requires $O(d^2)$ operations, instead of $O(2^{d^2})$ operations.

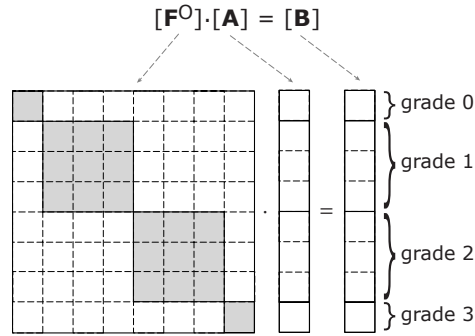


Figure 7: All elements in the outermorphism operator matrix $[\mathbf{F}^O]$ that are not gray are always 0.

Gaigen initializes the set of outermorphism matrices using images of all basis vectors e_i . These images specify what each basis vector should be transformed into, so we can immediately initialize the grade 1 matrix. Images of grade j basis elements are obtained by computing outer products of the appropriate basis elements of grade $j - 1 \dots 1$. The coordinates of these grade j basis elements can directly be copied into the appropriate column of the grade j outermorphism matrix. By applying this procedure recursively, we can fill up the matrices for grade $1 \dots d$. Applying an outermorphism to a scalar (grade 0) is not very useful, but if required the value of the scalar 'matrix' can be set by hand.

7 Fast dualization

Dualization with respect to pseudoscalar of the algebra can be implemented using this equation:

$$\mathbf{A}^* = \mathbf{A} \cdot \mathbf{I}^{-1} \quad (27)$$

However, a much more efficient implementation is possible when orthogonal basis vectors are used. In that case, dualization with respect to the pseudoscalar boils down to simply swapping and possibly negating coordinates. Efficient code for doing so can automatically be generated.

In Gaigen this is implemented by computing the inner product of each basis element \mathbf{E} with the inverse pseudoscalar \mathbf{I}^{-1} :

$$\mathbf{F} = \mathbf{E} \cdot \mathbf{I}^{-1} \quad (28)$$

The result of this product tells us to what basis element \mathbf{F} the coordinate that is currently referring to basis element \mathbf{E} should refer after dualization, and whether it should be negated or not. Once this has been computed for all coordinates, the source code for the fast dualization function can be generated.

8 Factoring Blades and Versors

The algorithm we use to compute the meet and join of blades, as described in section 9, requires that we can factor a blade (a homogeneous versor) into orthogonal vectors. That is, for an arbitrary n -dimensional blade B_n find a set of n vectors $v_1 \dots v_n$ such that $B_n = v_1 v_2 \dots v_n$.

We factor a blade B_n by repeatedly breaking it down until it is a scalar. During each step, we lower the dimension of the current blade B_i by removing an the inverse of an appropriate vector v_i from it:

$$B_{i-1} = B_i \cdot v_i^{-1} \quad (29)$$

If we repeat this operation n times we end up with n vectors and a scalar B_0 . So what we are doing is removing step by step dimensions from the subspace which B_n represents. The real problem is of course finding the appropriate vectors v_i to remove from the subspace.

The requirement for a vector v_i which can be removed from the blade B_i is that it is contained in B_i , that is $v_i \wedge B_i = 0$, and of course $v_i \neq 0$. We can find such vectors by projecting other vectors, of which we don't know for sure whether they are contained in B_i , onto B_i . A vector projected onto a blade is guaranteed to be contained in that blade. If we project a vector onto a blade and find that the result is not equal to 0, we have a candidate vector for the factorization. It is a *candidate*, because there are good candidates and bad candidates.

Suppose we decide to project a vector w onto the blade which is *nearly* orthogonal to it. Then the projected vector would be $w_p = (w \cdot B)B^{-1}$. The inner product used in the projection will give a result which is not equal to 0, but very small compared to the original w and B . This is bad for computational stability.

It happens [13] that there is simple and good method for selecting the basis vectors which are not (nearly) orthogonal to the blade. In Gaigen, a blade B_n of grade n is stored as coordinates referring to a n basis elements of grade n . The coordinates in the coordinate array (section 2) specify how much of each basis element is present in the blade. [13] proves that the basis vectors contained in the basis element that has the largest coordinate are not (nearly) orthogonal to the blade. Thus they are all good candidates for factorization.

So suppose a blade $B = 2e_1 \wedge e_3 \wedge e_4 + 3e_2 \wedge e_3 \wedge e_4 + 10e_1 \wedge e_3 \wedge e_5$, we see that the largest coordinate refers to $e_1 \wedge e_3 \wedge e_5$. We subsequently use e_1 , e_3 and e_5 in the algorithm described above to factor B : project the vector onto the blade, remove the projected vector from the blade, and repeat this with the next vector. We end up with a scalar, whose magnitude which can be equally distributed across all factors to make them of the same magnitude.

As a side note, the method described in this section can also be used to factor invertible versors. The highest non-empty grade (which is also a blade) is then used to pick the best basis vectors to project onto the versor. See [13] for details.

9 Meet and Join

The meet and join are two (non linear) products which compute the union (join) and intersection (meet) of subspaces represented by blades. An example of their use is the computation the intersection of lines, planes, points and volumes, all in a dimension independent way. What we mean by this is that traditional algorithms for find the union or intersection of different geometric primitives (lines, points, planes) into many cases. Such algorithms are error prone because they contain many different cases and exceptions. The meet and join provide a simple geometric operation for simplifying and unifying such algorithms and removing their many different cases.

The downside of this is that the meet and join as described here are less efficient than other, more direct methods. This may limit their use in time critical applications. Also, a sense of scaling is lost in the meet and join. Whereas the result of direct methods might be scaled with something like original magnitude of the input primitives and the cosine of the angle between them, meet and join have no sense of scale. The best one can do is make sure that the scale of the meet and join are dependent on each other.

All products computed during the meet and join algorithms (including the factorization described in the previous section) are performed using Euclidean metric. One reason we have to use Euclidean metric is for removing a vector from a blade using the inner product. Suppose the vector happens to be a null vector: the result of the inner product would be 0, while to vector would be orthogonal *in* the blade if one would use a Euclidean metric. Turning a non-Euclidean geometric algebra into a Euclidean one is called a LIFT and is described in more detail in [12].

As we will see, the computation of the meet can by done by computing the join (and the other way around). More specifically, Gaigen explicitly computes the join and bases the meet on it. Gaigen computes the join in a different way than GABLE [1]. GABLE contains rules that split the computation of the join into several cases, very much like traditional algorithms for computing intersections. This is feasible because GABLE implements the meet and join only for 3 dimensional geometric algebras. Gaigen can implement any low dimensional geometric algebra, and thus requires a more general algorithm. The algorithm works by first determining the grade of the join (by using a new product called the *delta product* [12]), and then constructing the join from the factored input blades.

9.1 The Delta Product

A major subproblem in computing the meet and join of subspaces (blades) is determining the grade of the result. If we are handed a grade 3 blade and a grade 2 blade in a 5 dimensional space, the outcome of the join can be a blade of grade 3, 4 or 5; the outcome of the meet can be a grade 0, 1 or 2 blade. The grade of the outcome depends on how (in)dependent the input blades are; i.e. how many dimensions are shared between the input blades. Once the grade of output blade is known, we can compute it using the algorithm described in the next section.

We compute the required grade of the meet and join of two blades A and B by computing the delta product $A\Delta B$ [12]. The delta product computes the

$$\begin{aligned}
\mathbf{A} &= \mathbf{a}_1 \wedge \mathbf{a}_2 \wedge \mathbf{a}_3 \\
\mathbf{B} &= \mathbf{b}_1 \wedge \mathbf{b}_2 \\
\mathbf{A} \Delta \mathbf{B} &= \mathbf{b}_1 \wedge \mathbf{a}_2 \wedge \mathbf{a}_3
\end{aligned}$$

Figure 8: Illustration of the delta product. The delta product computes the symmetric difference of those blades. (todo: proportional sign for '=' in last line)

symmetric difference of two subspaces represented by blades, hence the name.

Figure 8 shows an example: two blades \mathbf{A} and \mathbf{B} , factored in vectors \mathbf{a}_i and \mathbf{b}_i . The blades are factored such that shared dimensions are represented by equal vectors. In this example the blades share 1 dimension, $\mathbf{a}_1 = \mathbf{b}_2$. The delta product is proportional to the outer product of all vectors that \mathbf{A} and \mathbf{B} do not share in such a factorization. In this example: $\mathbf{A} \Delta \mathbf{B} = \mathbf{b}_1 \wedge \mathbf{a}_2 \wedge \mathbf{a}_3$ (todo: proportional sign for '=').

The grade of the delta product can be used to compute the grade of the meet and join: the grade of the join of \mathbf{A} and \mathbf{B} is

$$\text{grade}(\text{join}(\mathbf{A}, \mathbf{B})) = \frac{\text{grade}(\mathbf{A}) + \text{grade}(\mathbf{B}) + \text{grade}(\mathbf{D})}{2} \quad (30)$$

The motivation for this equation is that the sum of the grades of \mathbf{A} and \mathbf{B} counts the shared dimensions twice. By adding the grade of the delta product, we also count the non-shared dimensions twice. Thus dividing by two gives us the grade of the join.

The grade of the meet of \mathbf{A} and \mathbf{B} is

$$\text{grade}(\text{meet}(\mathbf{A}, \mathbf{B})) = \frac{\text{grade}(\mathbf{A}) + \text{grade}(\mathbf{B}) - \text{grade}(\mathbf{D})}{2} \quad (31)$$

This equation has a similar motivation. The sum of the grade of \mathbf{A} and \mathbf{B} counts the shared dimensions twice. By subtracting the grade of the delta product, we don't count the non-shared dimensions any more. Dividing this sum by 2 gives us the grade of the meet.

The delta product itself can be computed by taking the highest non empty grade of the *geometric* product $\mathbf{A} \mathbf{B}$ [12]. This may sound simple, but it is the crucial step in computing the grade of the meet and join. What if the highest grade part of a geometric product $\mathbf{A} \mathbf{B}$ is not equal to 0, but very small? Was this very small grade part caused by two *nearly* collinear factors in \mathbf{A} and \mathbf{B} or by floating point round off error? In the latter case, the very small grade part should be discarded, and we should look for the next highest grade part, thus changing the grade of the meet and join of \mathbf{A} and \mathbf{B} . This problem is the equivalent of problems that occur all over the place in traditional algorithms for computing the intersection of geometric primitives; that is, to decide whether two objects (lines, planes, etc) are collinear or not. This is usually dealt with by specifying some epsilon value ϵ : if the measure for non-collinearity of two objects falls below that ϵ , the objects are considered collinear, otherwise they are not.

9.2 Computing the Meet and Join

Once the required grade of the meet and join of subspace is known, our algorithm can compute the meet and join. Since the meet and join can be related by the equation

$$(\mathbf{A} \Delta \mathbf{B}) \cdot \text{join}(\mathbf{A}, \mathbf{B})^{-1} = \text{meet}(\mathbf{A}, \mathbf{B}) \quad (32)$$

we can suffice by computing the delta product and the join, from which we can compute the meet using a simple inner product.

Note that the scale of the meet and join can not be determined. This can be understood by looking at figure 8. We factored \mathbf{A} and \mathbf{B} and using this factorization, we can compute the delta product, the meet and the join. The scale of the individual factors however, is totally arbitrary. We may increase the scale of one factor, as long as we decrease the scale of another factor proportionately. This will change the scale of the delta product, meet and join. Or to look at it in another way: we are using blades to represent subspaces. The same subspace can be represented by an infinite number of blades, which are all related to each other by a scale factor. The blades \mathbf{C} , $2\mathbf{C}$ and $-\mathbf{C}$ all represent the same subspace. So the best thing we can do is to make the meet and join proportional to each other, e.g.:

$$((\mathbf{A} \Delta \mathbf{B}) \cdot \text{join}(\mathbf{A}, \mathbf{B})^{-1}) \cdot \text{meet}(\mathbf{A}, \mathbf{B})^{-1} = 1 \quad (33)$$

Computing the meet and join in this way also has the advantage that computing either the meet or the join is enough to compute the other. Thus we need only one algorithm. We choose to implement the join and base the meet on it.

To compute the $\text{join}(\mathbf{A}, \mathbf{B})$ we can construct *any* blade which is non-zero, and contains both the subspaces which blades \mathbf{A} and \mathbf{B} represent. This blade could be factored as:

$$\text{join}(\mathbf{A}, \mathbf{B}) = \alpha \mathbf{A}' \wedge \mathbf{C} \wedge \mathbf{B}' \quad (34)$$

where α is any scalar not equal to 0, \mathbf{C} is a blade which is contained in both \mathbf{A} and \mathbf{B} ($\mathbf{C} \wedge \mathbf{A} = \mathbf{C} \wedge \mathbf{B} = 0$), $\mathbf{A}' = \mathbf{A} \cdot \mathbf{C}^{-1}$ and $\mathbf{B}' = \mathbf{C}^{-1} \cdot \mathbf{B}$. Because $\mathbf{A} = \mathbf{A}' \mathbf{C} = \mathbf{A} \mathbf{C}^{-1} \mathbf{C}$, all we have to do is find a blade which is proportional to \mathbf{B}' .

Let us assume \mathbf{A} is the highest grade blade, and \mathbf{B} is of lower or equal grade (we swap the labels of the blades when this is not true). We could take \mathbf{A} and wedge a blade \mathbf{D} to it such that it contains *both* blades \mathbf{A} and \mathbf{B} . So $\text{join}(\mathbf{A}, \mathbf{B}) = \mathbf{A} \wedge \mathbf{D}$.

Since we already know from the delta product $\mathbf{A} \Delta \mathbf{B}$ what the grade of the result $\text{join}(\mathbf{A}, \mathbf{B})$ should be, we know that $\text{grade}(\mathbf{D}) = \text{grade}(\text{join}(\mathbf{A}, \mathbf{B})) - \text{grade}(\mathbf{A})$. We have devised two algorithms to find this blade \mathbf{D} .

The first algorithm factors the lower grade blade \mathbf{B} and then tries to wedge combinations of these factors to \mathbf{A} . We require $n = \text{grade}(\mathbf{D})$ factors to get a blade of the required grade. If we select the right factors $\mathbf{b}_1, \mathbf{b}_2 \dots \mathbf{b}_n$, then the result $\mathbf{A} \wedge \mathbf{b}_1 \wedge \mathbf{b}_2 \wedge \dots \wedge \mathbf{b}_n$ should not be equal 0. The problem is thus to find the correct factors from \mathbf{B} to wedge to \mathbf{A} .

We currently solve this problem by trying all combinations of factors from \mathbf{B} and keeping the blade which has the largest euclidean norm. This requires 'take grade(B) out of grade(D)' tries. (todo: combinatoriek).

The second algorithm projects basis elements \mathbf{e}_b of grade(D) onto \mathbf{B} , thus making sure that they are contained in \mathbf{B} . It then wedges them to \mathbf{A} . If the

Layer 2: interface	C++, object oriented, operator overloading, user extensible, high level algorithms.
Layer 1: glue and structure	C++, object oriented, memory management glues together the layer 0 functions, lookup tables, low level algorithms, profiling
Layer 0: computational	Low level computational functions, generated by a .opt-compiler in C or (SSE-, 3DNow!-, AltiVec-) assembler. Implements multiplication, addition, reverse, copy, etc.

Figure 9: The three layers of Gaigen.

result $\mathbf{A} \wedge ((\mathbf{e}_b \cdot \mathbf{B})\mathbf{B}^{-1})$ is not equal to 0, we have found a blade which is representative for $\text{join}(\mathbf{A}, \mathbf{B})$. The problem is to find the correct basis element \mathbf{e}_b to project onto \mathbf{B} . We currently solve this problem by trying all possible basis elements of $\text{grade } \text{grade}(\mathbf{D})$ and using the one which resulted in the blade with the largest euclidean norm. (todo: combinatoriek).

10 Implementation

Gaigen source code is divided into three layers, as can be seen in figure 9. Layer 0 is partly generated from a .opt file, which is in turn generated by Gaigen. A .opt file describes at a very low level how to multiply the coordinates of multivectors for each product. It also specifies which specific product/multivector combinations have to be optimized. The following is an excerpt from a .opt file which specifies how to implement an optimized version of the geometric product of a 3D vector and a 3D rotor:

```
optimize e3ga_opt_02_gp_05
c[0] = + a[0] * b[0] - a[1] * b[1] - a[2] * b[2]
c[1] = + a[1] * b[0] + a[0] * b[1] - a[2] * b[3]
c[2] = + a[2] * b[0] + a[0] * b[2] + a[1] * b[3]
c[3] = + a[2] * b[1] - a[1] * b[2] + a[0] * b[3]
```

The .opt file contains such specifications for every product and for every optimized product/multivector combination. The .opt file should be compiled by a separate program. Currently we have only implemented a .opt→.c compiler, but we plan to implement a .opt→SSE assembler and a .opt→3DNow! compiler, to make use of the SIMD instructions the mainstream processors produced by Intel and AMD offer. Users can create their own .opt compiler to optimize Gaigen to the specific capabilities of their platform. The other part of layer 0 is a collection of low level functions which perform operations such as addition, reversion, grade involution and copying of multivectors. These can also be replaced with versions optimized for a specific platform.

The main tasks of layer 1, which is also generated by Gaigen, are to glue the layer 0 functions together, to supply a structure or class in which to store the coordinates and grade usage information, and to provide low level algorithms or operations such as dualization, projection, meet and join, inverse, and outermorphism.

Layer 2 is the interface between the user and the low level Gaigen source code, and provides an abstraction from the implementation details. It defines the operator symbols (such as $*$, $-$, $+$, $/$, $\&$, $!$ and \ll). For all these operations normal functions are also available (such as `gp`, `subtract`, `add`, `igp`, `meet`, `dual` and `lcont`). For operations for which there is no operator symbol, normal functions are provided (such as `hip()` for 'Hestenes Inner Product' or `gradeInvolution()`).

Gaigen was divided into three layers with the following motivations in mind. First of all a layer was required to contain optimized code which would be generated in platform specific (assembly) code (layer 0). Secondly, we wanted a top layer which could be modified and adapted by the user (layer 2), because we don't want to enforce our preferences (e.g. which operator symbols are used for what purpose) upon users. Also the user should be able to add often used algorithms to the Gaigen source code. Because we did not want the user to become involved with the implementation details of layer 0, layer 1 was required to hide these.

Of course, layer 1 and layer 0 can be modified by the user too, but this is not recommended unless the user has knowledge of the internal operation of Gaigen. Layer 0 should only be edited or compiled by a compiler to optimize it for a specific platform. Most of the layer 1 is generated from a template file. This template file can be modified as well.

11 Performance

In this section we present measurements of the performance of Gaigen. The goal is to show how Gaigen compares to other methods of implementing geometric algebras, to show the impact on performance of various options that Gaigen offers, and to show how Gaigen compares to standard implementation of linear algebra for an equivalent model.

Many of the benchmarks were done in a full application. We prefer a full application over synthetic benchmarks because it is hard to pick a representative set of operations that the synthetic benchmark should cover. E.g. simply computing the geometric product of two fully general multivector may be the most expensive product to compute, but will rarely if ever be used in geometry. We specifically wrote a simple recursive ray tracer with the purpose to benchmark Gaigen and some geometric algebra models in general because a ray tracer uses a wide selection of geometric operations.

11.1 Performance compared to GABLE

GABLE is an educational geometric algebra package implemented in Matlab. It is described in [1]. In [15], GABLE was used to implement an algorithm for finding singularities in a vector field, but performance was too low. Later,

the same algorithm was implemented using Gaigen instead of GABLE, and performance went up by a factor of approximately 6000.

11.2 Ray tracer performance

We implemented a recursive ray tracer algorithm multiple times, each time using a different algebra to implement the geometry [16]. We did this to compare the performance and elegance of various methods of doing 3D Euclidean geometry in a practical application, and to compare the performance of Gaigen to other implementations of linear and geometric algebras.

The algebras used are 3D linear algebra, 4D linear algebra (homogeneous coordinates and Plücker coordinates), 3D geometric algebra, 4D geometric algebra (homogeneous model), and 5D geometric algebra (conformal model). The 3D LA and 3D GA models of geometry are each others equivalents, as are the 4D LA and 4D GA models. In this context, by *equivalent* we mean that almost identical operations (at the computational level) are used to implement geometric operations.

We wrote 'standard' optimized implementations of linear algebra by hand. To implement the geometric algebras we used both Gaigen and CLU [17]. This was done to compare the performance of optimized Gaigen code to CLU. CLU is a geometric algebra implementation with emphasis on functionality rather than performance. Of course, we enabled all speed optimizations we could in Gaigen.

The results are in figure 10. The render times are for rendering one small 320×240 pixel image, from a scene containing about 8000 triangles. There are two columns with render times, each one for a different setting of the ray tracing program (performance can be measured with or without time spent on line-BSP intersections). There is also a column that lists the size of the executable program and a column listing the amount of memory used at run time while rendering the image.

We found that in this application Gaigen is $2 \times$ to $3 \times$ slower than our standard linear algebra implementation in equivalent models. The 5D GA conformal model performs about $2 \times$ slower than the 3D GA model. We also found that Gaigen is about $30 \times$ to $90 \times$ faster than CLU.

11.3 Coordinate memory allocation and floating point type

To investigate the performance impact of the different coordinate memory (re-) allocation schemes that were described in section 2.4, we benchmarked the ray tracing application using different settings for these options. We used the Gaigen 3D GA and Gaigen 5D GA versions of the ray tracer as basis. We varied the setting of the floating point type (32 bit precision *floats* and 64 bit precision *doubles*), and the memory (re-)allocation algorithm.

Results are shown in figure 11. We found the obvious result that floats are faster than doubles, and require less memory. We also found that the maximum parity pure (maxpp) memory allocation method is usually the most efficient method, but that memory usage can be reduced by using the *tight* memory allocation method. This reduction can be quite significant when higher dimensional (i.e. 5D) algebras and/or doubles are used. The *balanced* memory

model	implementation	full render time	render time w.o. BSP	executable size	run time memory usage
3D LA	standard	1.00× (23.3s)	1.00× (0.99s)	52KB	6.2MB
4D LA	standard	1.05×	1.22×	56KB	6.4MB
3D GA	Gaigen	2.56×	1.86×	64KB	6.7MB
4D GA	Gaigen	2.97×	2.62×	72KB	7.7MB
5D GA	Gaigen	5.71×	4.58×	100KB	9.9MB
3D GA	CLU	129×	72.0×	164KB	12.6MB
4D GA	CLU	164×	97.1×	176KB	14.7MB
5D GA	CLU	482×	178×	188KB	19.0MB

Figure 10: Performance benchmarks run on a Pentium III 700 MHz notebook, with 256 MB memory, running Windows 2000. Programs were compiled using Visual C++ 6.0. All support libraries, such as fltk, libpng and libz were linked dynamically to get the executable size as small as possible. Run time memory usage was measured using the task manager.

allocation algorithm is practically useless, at least in its current implementation. It is slower than the *tight* method, and in most cases uses more memory than the *maximum parity pure* method. A surprising result was that in some cases the *tight* method can be faster than the *maximum parity pure* method, i.e. with the 5D GA with doubles as floating point type. This is probably related to memory caching in the CPU.

11.4 Optimizations and dispatching methods

We also wanted to know what dispatching method (described in section 4.1) performs best in practice. The dispatching method handles the task of getting from the product function call to the actual (optimized) function that implements the product.

We took the Gaigen 3D GA and Gaigen 5D GA ray tracer implementations and used different settings for the dispatching option of the algebra. We also disabled optimizations entirely, to see what impact that would have on performance.

The results are figure 12. The *if else* dispatching method is most efficient, closely followed by the *switch* method. The *lookup table method* is not to be recommended, although initially it was the only dispatching method available in Gaigen. Specifying no (or useless) optimizations for the algebra results in poor performance, more than 3× lower than with optimizations.

11.5 Theoretical performance without dispatching

Dispatching and grade part usage checking currently form a major bottleneck in the performance of Gaigen. It is required not only for the products, but to a lesser extent also for much simpler functions like addition, reversion and fast dualization: every time some operation has to be performed on a multivector, Gaigen has to check what grade parts are in use.

model	floating point type	memory (re-) allocation method	render time	run time memory usage
3D GA	float	tight	89.5s	6.7MB
3D GA	float	balanced	100.7s	7.0MB
3D GA	float	maxpp	59.4s	6.7MB
3D GA	float	max	61.9s	7.8MB
3D GA	double	tight	100.8s	8.2MB
3D GA	double	balanced	111.6s	8.5MB
3D GA	double	maxpp	85.3.s	8.7MB
3D GA	double	max	86.7s	10.7MB
5D GA	float	tight	150.0s	7.7MB
5D GA	float	balanced	162.1s	12.5MB
5D GA	float	maxpp	133.2s	9.9MB
5D GA	float	max	135.9s	14.0MB
5D GA	double	tight	176.2s	9.8MB
5D GA	double	balanced	189.8s	14.2MB
5D GA	double	maxpp	234.1s	14.7MB
5D GA	double	max	236.6s	23.0MB

Figure 11: Performance impact of the memory (re-)allocation methods. All benchmarks were run under the same conditions as those in figure 10, with the same optimizations and dispatching method; only the floating point type and the allocation method were varied.

model	dispatching method	render time
3D GA	if else	59.4s
3D GA	switch	59.6s
3D GA	lookup table	62.8s
3D GA	no optimizations	215.7s
5D GA	if else	132.6s
5D GA	switch	135.0s
5D GA	lookup table	137.0s
5D GA	no optimizations	442.0s

Figure 12: Performance of various dispatching methods, and without product optimizations. All benchmarks were run under the same conditions as those in figure 10, with identical optimizations (unless optimizations were turned off) and floating point type and memory allocation method; only the dispatching method was varied.

model	operation	dispatching method	time
3D GA	$v_1 \cdot v_2$	if else	0.82s
3D GA	$v_1 \cdot v_2$	hard coded	0.12s
3D GA	$r v r^{-1}$	if else	2.53s
3D GA	$r v r^{-1}$	hard coded	0.54s
5D GA	$v_1 \cdot v_2$	if else	1.03s
5D GA	$v_1 \cdot v_2$	hard coded	0.12s
5D GA	$r v r^{-1}$	if else	15.9s
5D GA	$r v r^{-1}$	hard coded	2.54s

Figure 13: Theoretical performance without dispatching. The times are for executing the given operation 10,000,000 times.

The grade part usage checks are necessary because Gaigen uses a single datatype to store all types of multivectors. This is nice for generality, but bad for performance. The compiler can not decide at compile-time what type of multivectors are passed to a function and can thus not make the dispatching decision at compile-time. The processor has to do so at run time, which lowers performance, especially compared to linear algebras (where people are used to having only a few, hard datatypes).

If we would introduce datatypes for all types of multivectors we want to use (e.g. 1-vectors, bivectors, rotors, versors) the compiler could make dispatching decisions at compile-time. To see what the performance increase would be if we implemented this idea, we did some artificial benchmarks. We implemented two operations in two algebras. The algebras are a 3D geometric algebra, and the 5D geometric algebra that is used to implement the conformal model in the ray tracer. The operations are a simple inner product of vectors ($v_1 \cdot v_2$) and a more complicated versor product ($r v r^{-1}$). In the 5D algebra, the versor r is a product of a translator and a rotor and thus it occupies grade parts 0, 2 and 4; in the 3D algebra the versor is a rotor. In one set of benchmarks we used the *if else* dispatching method as baseline, in the other we *hard coded* (by hand) the function call to the required optimized function.

The results are in figure 13. we found that for the simple products (requiring less multiplications/additions), the gain can be in the order of $10\times$. The more complicated products gain performance by a factor of about 5. Note that these are raw performance benchmarks, and not full application benchmarks. In full application benchmarks the contribution of the performance gain related to the GA implementation is blurred a little because time is also spent on other computations.

11.6 Best settings for performance

We conclude that Gaigen performs best with memory allocation method set to *maxpp*, *floats* as floating point type, the *if else* dispatching method and optimizations made for all product/multivector combinations in use. When using *doubles* in high dimensional algebras, the *tight* memory allocation method may perform better.

12 Discussion, conclusion

12.1 Discussion

The way Gaigen implements geometric algebras seems feasible. We have worked with it in practice for over a year and found it usable. The process of optimizing your algebra for a specific application is slightly annoying but doesn't take much time in practice, especially with the option to directly read profiles back into the Gaigen UI and letting Gaigen do the optimizations automatically.

We see no other general way to implement low dimensional geometric algebras as efficiently as current linear algebra implementations than to take at least a partial code-generation approach. Implementing the whole algebra using C++ templates and the like (i.e. like CLU) causes too much overhead. Writing every algebra or even its product code by hand is too tedious. Of course, Gaigen is quite extreme in that almost all code is generated, but the products will always have to be generated and turned into code. However, this process could be done at run-time (dynamic code generation), instead of at compile-time like Gaigen currently does.

A lot of improvements are still possible before we reach the maximum performance software-only implementations of geometric algebra could achieve. Although Gaigen is a huge step forward compared to the performance of GABLE ($> 1000\times$ faster), and an order of magnitude step forward compared to CLU ($> 20\times$ faster), we still have a factor of 3 (in full application benchmarks) to 10 (in raw benchmarks) to go before we obtain the performance of equivalent linear algebra methods. We are looking at a number of improvements to reach this goal.

As explained in section 11.5, getting rid of the dispatching and grade part usage checks would be a big gain. As shown by the benchmark, the dispatching in products can lead to a raw performance drop up to a factor of 10. We intend to remove the dispatching by introducing to option to generate data types for specific multivector types such as blades and versors. The challenge here will be maintain full generality (i.e. if you want to only program using 'general multivectors', you should be able to), while going for maximum performance. The introduction of the new data types should be as transparent to the user as possible.

The new data types would also be more efficient with respect to memory usage. Currently, even the *tight* memory allocation method wastes memory: we have to keep track of grade part and memory usage and keep a pointer to the array holding the coordinates. Depending on the processor architecture, this wastes 6 to 12 bytes for each multivector (the three floats holding the coordinates of an ordinary 3D vector require 12 bytes for storage, so in that case, memory wastage could be as high as 100%).

Gaigen splits multivectors into grade parts and keeps track of what grade parts are in use (not equal to 0). This works well for low dimensional algebras. But in the 5D conformal model, we already notice that some coordinates are always 0 for certain objects. E.g. lines and encoded by trivectors in the conformal model, which leads to the usage of 10 coordinates. However, only 6 of those coordinates are used, the others are always 0. A circle in the conformal model is also encoded by a trivector, but *does* use all 10 coordinates of grade 3. This suggests that it might be useful to split the grade parts further into sub grade

parts. I.e. in the conformal model we might want to separate the coordinates having to do with 'infinity' (e_∞) from those that don't have to do with e_∞ .

An easy to implement optimization is the dual version of certain products. We have noticed that we often need to compute products like $A^* \cdot B$. To implement this, currently one first has to compute the dual of A and then compute the inner product. It would be quite simple to do the dualization 'on the fly', i.e. to generate code for a dual inner product that performs the dualization internally. Of course, this would only work for dualization with respect to the full space, just like the fast dualization method described in section 7.

Although we make mention of the possibility to generate optimized code for floating point SIMD instruction sets, we have had no success yet at implementing this in practice for SSE (the Pentium 3 SIMD instruction set). The SSE instruction set just seems too linear algebra/signal processing minded to be of any use for this purpose. Most geometric algebra product (except the outermorphism) require the ability to arbitrary swizzle and negate elements in the coordinate array, and we weren't able to implement this efficiently: ordinary floating point code generated by a standard compiler was faster. We have not yet verified whether the same is true for other processors (AMD, Motorola/Apple). Ironically, modern 3D graphics processing units *do* have arbitrary and costless swizzle and negate option built in.

Something we have not yet considered but might be a useful feature is the interoperability between geometric algebras. Suppose someone want to use both a 3D Euclidean and a 5D conformal geometric algebra in the same application. Although this is entirely possible with Gaigen, there are no functions provided to 'translate' multivectors from one algebra to the other. One has to go to the coordinate level to do this. It would be nice if Gaigen could generate such translation functions, but what these translation functions should do exactly depends of course on the interpretation the user assigns to the multivectors from the respective algebras.

The three layer design described in section 10 might be nice from a design point of view, but we could do fine with a two layer approach. In practice we never changed layer 2 (the high level C++ interface) after it was finished. The idea was that one could easily insert new algorithms into this top layer, but we always just put them in separate functions. So a two layer design, in which one layer with optimized functions and the other is generated from a template file would be sufficient.

13 Conclusion

We have implemented low dimensional geometric algebras in C++ using a code generator. To obtain an implementation of a specific geometric algebra, the user only has to supply the specifications of the algebra. Gaigen then automatically generates the source code, including optimized product function. We have successfully used this source code to implement a number of applications, including a recursive ray tracer.

We have reduced memory usage and increased performance by tracking which grade parts of multivectors are in use. This grade part usage information is also used to enable the usage of optimized product functions.

Extensive benchmarks show that this results in an geometric algebra implementation that is less than an order of magnitude slower than equivalent a standard linear algebra implementation. We have suggested and experimented with methods to overcome this difference. We hope these improvements will close the gap between GA and LA performance, which might lead to faster adaption of GA by mainstream programmers.

We have shown how to implement a number of geometric algebra functions such as meet and join, blade and versor factorization, fast dualization and inversion.

References

[0] Todo: fix all references...

[1] S. Mann, L. Dorst, T Bouma, *The Making of GABLE: A Geometric Algebra Learning Environment in Matlab*, in Geometric Algebra with Applications in Science and Engineering, E Corrochano and G Sobczyk (eds), Birkhauser, 2001, pg 491-511. Also available at <ftp://cs-archive.uwaterloo.ca/cs-archive/CS-99-27/> and <http://carol.science.uva.nl/~leo/clifford/gable.html>, 1999

[2] L. Dorst, S. Mann, T. Bouma. *GABLE: A matlab tutorial for geometric algebra*. <http://carol.science.uva.nl/~leo/clifford/gable.html>, 1999

[3] C. Doran and A. Lasenby. *Physical applications of geometric algebra*. <http://www.mrao.cam.ac.uk/~clifford/ptIIIcourse/>, 1999

[4] H. Li, D. Hestenes, A. Rockwood. *Generalized Homogeneous Coordinates for Computational Geometry. ???*.

[5] D. Hestenes. *New Foundations for Classical Mechanics, Second Edition*. Reidel, 1999.

[6] D. Fontijne. *Gaijen user manual*. Does not exist yet...

[7] Advanced Micro Devices. *3DNow!tm Technology Manual*. Currently available at <http://www.amd.com/K6/k6docs/pdf/21928.pdf>

[8] Intel. *Pentium 4 processor manuals*. Currently available at <http://developer.intel.com/design/Pentium4/manuals/>

[9] Apple. *Apple's AltiVec homepage* Currently at <http://developer.apple.com/hardware/ve/index.html>

[10] P. Lounesto. *Clifford Algebras and Spinors* London Mathematical Society Lecture Note Series 239, Cambridge University Press, 1997.

[11] J. Lasenby, A.X.S. Stevenson *Using geometric algebra for optical motion capture* to appear in in Applied Clifford Algebras in Computer Science and Engineering, Ed. E.Bayro-Corrochano and G. Sobczyk. Birkhauser 2000.

[12] T. Bouma *About the delta product* Where: ask Tim.... / online

[13] T. Bouma *About versors* Where: ask Tim.... / online

- [14] L. Dorst and S. Mann. *Geometric algebra: a computation framework for geometrical application, Part I and II*. IEEE Computer Graphics and Applications, Vol. 22, No. 3, May/June 2002, and Vol. 22, No. 4, July/August 2002.
- [15] S. Mann and A. Rockwood. *Computing Singularities of 3D Vector Fields with Geometric Algebra*. In: IEEE Visualisation 2002.
- [16] D. Fontijne and L. Dorst. *Performance and elegance of five models of 3D Euclidean geometry in a ray tracing application 2002*. Submitted to IEEE Computer Graphics and Applications.
- [17] C. Perwass. *The CLU project 2002*. Cookville conference.