

Gaigen 2: a Geometric Algebra Implementation Generator

Daniel Fontijne

University of Amsterdam
fontijne@science.uva.nl

Abstract

Geometric Algebra (GA) is an algebra that encodes geometry much better than standard techniques, which are mainly based on linear algebra with various extensions. Compared to standard techniques, GA has clearer semantics and a richer, more consistent language. This expresses itself, among others, in a much greater genericity of functions over the algebra. Exploiting this genericity efficiently is a problem that can be solved through generative programming.

This paper describes our Geometric Algebra Implementation Generator *Gaigen 2*. *Gaigen 2* synthesizes highly efficient GA implementations from the specification of the algebra. Functions over such algebras can be defined in a high-level coordinate-free domain-specific language, and *Gaigen 2* transforms these functions into low-level coordinate-based code. This code can be emitted in any target language through a custom back-end. Benchmarks of our implementation show that the combination of GA and *Gaigen 2* can rival the performance of standard geometry techniques, despite the greater abstraction and genericity of GA.

To obtain this high performance, *Gaigen 2* must adapt the generated code to the program that links to it. This is done via a profiling feedback loop. While running, the generated code makes a connection to the code generator. The generated code sends information about functions that should be optimized. The code generator registers this information and sends back new type information. After the program terminates, the code is regenerated according to the recorded profile. This profiling feedback technique may also be useful to implement other types of algebras.

Categories and Subject Descriptors G.4 [Mathematical Software]: Efficiency

General Terms Performance, Design, Languages.

Keywords Geometric algebra, conformal model, object-oriented, profiling, program transformation, synthesis from specification.

1. Introduction

Geometric algebra is the geometric re-interpretation of Clifford Algebra. It can be used to *model* various geometries, such as space-time, conformal and hyperbolic geometry. One of the distinctive properties of geometric algebra is the genericity of its functions: functions over the algebra can often be applied to many different types of objects without modification. For example in the conformal

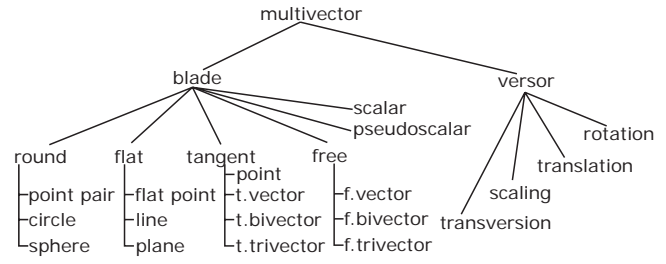


Figure 1. Classification of the blades and versors of the conformal model of 3D geometry. Note the similarity to an object-oriented inheritance tree. This figure is a simplification: invertible blades are versors too, and versors can be multiplied (e.g., to form translation-rotation versors). The strange location of the *point* in the tree can be justified by considering it to be a ‘tangent scalar’, i.e., a scalar with a position. Note that we use the term ‘object-oriented’ as an analogy, and not in the strict programming language sense. I.e., in our *Gaigen 2* implementation, the leafs of the tree (circle, plane, etc) are not subclasses of the multivector class; the knots (blade, versor, etc) in the middle of the tree are not even implemented.

mal model, the equation to rotate a vector is no different than the equation to rotate a circle or any other object, for that matter. We consider geometric algebra to be the high-level ‘object-oriented’ language for encoding geometry, whereas – in this context – linear algebra is more akin to assembly language. See Figure 1 to get an idea of how geometric algebra is object-oriented.

GA can be applied to any area of computer science that uses geometry, such as computer graphics, computer vision, robotics and physics simulation. Geometric algebra unifies many different formalisms that are used in these areas. Complex numbers, quaternions and Plücker coordinates are all elements of geometric algebra. These are currently considered extensions, tricks or separate algebras on their own. But by studying GA, many well-known pieces of the geometry puzzle suddenly fit together to form a consistent whole. While this is astounding on a mathematical level, it also has practical applications: connecting various components is simpler when all components speak the same (geometric) language.

However, when someone is first introduced to geometric algebra, the reply is more often than not a sceptic “*Why?*”. Mastering geometric algebra can take up to a year, and it seems to add little to standard techniques which the person already mastered. The phrase “old wine in new bottles” often arises. This fact, combined with the fact that most members of the GPCE community are unlikely to be familiar with GA, lead us to write this paper with a rather unconventional layout:

- First, we motivate why one would want to study GA in general. We also motivate why GA is an interesting problem domain for the (generative) programming community.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GPCE’06 October 22–26, 2006, Portland, Oregon, USA.
Copyright © 2006 ACM 1-59593-237-2/06/0010...\$5.00.

- Next we give an informal introduction of geometric algebra at the level required to understand the description of Gaigen 2.
- This is followed by a introduction of GA implementation, again at the level required to understand the Gaigen 2 description. We also discuss several existing GA implementations, focussing on their performance.
- The heart of the paper is the description of our geometric algebra implementation generator Gaigen 2.
- We conclude with several benchmarks which allow us to compare the performance of Gaigen 2 to various alternatives for implementing geometry.

2. Why Geometric Algebra?

In the introduction we compared geometric algebra to an object-oriented language and linear algebra to assembly language. This comparison holds on multiple levels:

1. Non-optimized GA implementations are slower than standard LA implementations. The higher level of abstraction of GA comes at the price of lower performance. However – as with programming languages – a good ‘optimizing GA compiler’ can solve this problem for the most part. Our Gaigen 2 is such a compiler.
2. GA-based programs are easier to implement and maintain, because the geometry can be encoded with shorter, simpler, and more generic expressions. Also, it is easier to visualize the geometry. Just as with assembler vs OOP, the gain is in how *quickly* and *easily* you can program something, not in *what* you can program.
3. In LA, you literally have to ‘assemble’ representations of objects from a very limited set of primitives (mostly vectors, matrices and scalars) and operate on them with a limited set of operations (dot product, cross product, matrix multiplication). This makes LA based geometry very low level. GA on the other hand offers a much richer set of primitives and operations.

We describe our approach to the first point in Section 7. Let us illustrate the two other points with an example:

The conformal model is an application of GA which is very well suited for encoding Euclidean geometry. We describe it in some more detail in Section 4, and use it as a source of examples throughout this paper. Figure 1 shows the primitives that can be directly represented by a *multivector* in the conformal model of 3D geometry [1]. The multivector is the general element of computation of the algebra. The figure shows how the multivectors in the conformal model can be classified. The first distinction is between *blades* (objects) and *versors* (operations). The blades can be further classified into the scalars and pseudoscalars, and four groups:

- The round blades: objects such as circles and spheres.
- The flat blades: objects such as lines and planes.
- The tangent blades: directional elements that also have a position. In 3D, there are 3 variants: tangent vectors, tangent *bivectors* (‘surface elements’), and tangent *trivectors* (‘volumes’).
- The free blades: directional elements that have a no position (they are translationally invariant).

The versors represent transformations such as rotation and translation. The products of these versors represent combinations of these transformations, but these are not shown in the figure.

The large ‘vocabulary’ of conformal GA is very useful when implementing geometry. For example, to represent the concept ‘ray’, a tangent vector can be used. The tangent vector type is automatically present in any implementation of the conformal model, so

no special effort is required to implement it. On the other hand, in a standard LA based geometry implementation we would have to create a class to represent it, e.g.:

```
class Ray {
    vector position;
    vector direction
};
```

Suppose we also need the concept ‘circle’, which can also be directly represented by GA. In LA we have to write a class like:

```
class Circle {
    vector position;
    vector normal;
    float radius;
};
```

When using LA to implement the concepts in the tree in Figure 1, we would have to write a class for almost all of them. Now suppose we also want to be able to transform (translate, rotate, scale, reflect) them. The function that does this in GA is very simple:

```
multivector applyVersor(versor X, multivector A) {
    return X * A / X;
}
```

This function can apply any versor to any multivector. Hence it can translate, rotate, scale, reflect all of the concepts listed in the tree in Figure 1. The special properties of the multivector – such as the translational invariance of the free vectors – are automatically taken into account by the algebra. When using LA, the transformation is specified using a matrix. Special code would have to be written to apply the matrix to each object. For example, for the ray:

```
ray transform(matrix M, Ray R) {
    Ray result;
    result.position = transformPoint(M, R.position);
    result.direction = transformDirection(M, R.direction);
    return result;
}
```

Here we assume a number of `transformX()` functions have already been provided which take into account the special properties of ‘X’. The code to transform a circle would be:

```
Circle transform(matrix M, Circle C) {
    Circle result;
    Circle.position = transformPoint(M, C.position);
    Circle.normal = transformNormal(M, C.direction);
    Circle.radius = transformSize(M, C.radius);
    return result;
}
```

This type of code would have to be written for each member of the tree. Now suppose we also want to be able to perform intersection or projection or spanning operations with the members of the tree, and you understand what a tremendous redundant implementation effort it would be to use LA for this purpose, when each operations can be encoded with a simple GA expression.

Of course, there are limitations to what geometric algebra can represent. For example, no algebra has yet been found that can directly represent triangles, or splines. The representation of such objects still has to be built from more primitive objects and relations, just as with LA. Examples of what GA can handle are (but are not limited to) manipulations on images, conics and Euclidean, space-time, hyperbolic, conformal and projective geometry.

Note that even though GA can encode many functions generically, it is not straightforward to get this idea to work efficiently. Without proper optimizations, GA-based geometry can easily be two orders of magnitude slower than similar LA-based geometry. See the benchmarks in Section 8.1. This is the problem we solve with Gaigen 2.

This problem is also one of the reasons GA may be interesting to the generative programming community:

- GA exposes limitations of (main-stream) programming languages, more so than LA and tensor algebra do.
- Generative programming can overcome these limitations.
- GA is ‘object-oriented’ by nature.
- GA is semantically much clearer than LA. E.g., LA has only one type of ‘vector’ that gets re-used over and over again in different roles (point, direction, normal, translation). As expected, this often leads to programming errors. GA on the other hand offers a separate type for each of these concepts. The clearer semantics of GA may be useful for automatically proving the correctness of geometric programs.

3. Introduction to GA

There are many introductions to GA [2, 3], but to enable stand-alone reading of this paper, we include a concise GA introduction.

The elements of a geometric algebra are called *multivectors*. There are two useful types of multivectors: the *blades* and the *versors*. A blade is an *outer product* of vectors. A versor is a *geometric product* of vectors. There are also multivectors that are neither blades nor versors, but these have little use in GA because they are geometrically meaningless. The most important products are:

- the outer product, used for spanning,
- the inner product, used to compute incidence relationships like distance and angle, and
- the geometric product, used to create and apply transformations.

For each product we list the typical usage, but of course there are other uses for each product.

Given two vectors \mathbf{a} and \mathbf{b} , the outer product – denoted by \wedge – computes the blade which the vectors span:

$$\mathbf{B} = \mathbf{a} \wedge \mathbf{b}.$$

\mathbf{B} is a *2-blade* or *bivector*, where the ‘2’ refers to the *grade* (dimensionality) of the blade. Think of \mathbf{B} as an oriented, scaled, surface area through the origin, as in Figure 2a. All blades are oriented, and orientation is carried on through almost every product in GA.

Vectors are 1-blades and it is convenient to call the scalar elements of the algebra ‘0-blades’. A 3-blade or trivector can be constructed by *wedging* three vectors together:

$$\mathbf{C} = \mathbf{a} \wedge \mathbf{b} \wedge \mathbf{c}.$$

Think of \mathbf{C} as an oriented volume at the origin (see Figure 2b). The outer product is anti-commutative: *reversing* the order of the vectors can flip the orientation of the blade:

$$\text{reverse}(\mathbf{a} \wedge \mathbf{b}) = \widetilde{\mathbf{a} \wedge \mathbf{b}} = \mathbf{b} \wedge \mathbf{a} = -\mathbf{a} \wedge \mathbf{b}.$$

Computing the outer product of linearly dependent vectors yields $\mathbf{0}$; hence in 3D space, there are no 4-blades.

Blades are used to represent the *objects* of the geometry that you want to model. In their most basic interpretation they represent oriented, scaled, subspaces through the origin, but in the conformal model blades are interpreted as objects like lines, circles and planes.

In geometric algebra, the inner product works on all combinations of blades, instead of just on vectors. For example, the inner product of a vector and a 2-blade is:

$$\mathbf{d} = \mathbf{c} \cdot (\mathbf{a} \wedge \mathbf{b}) = (\mathbf{c} \cdot \mathbf{a}) \mathbf{b} - (\mathbf{c} \cdot \mathbf{b}) \mathbf{a},$$

see Figure 2c. The inner product is used to compute incidence relationships, such as distance, angle and intersection.

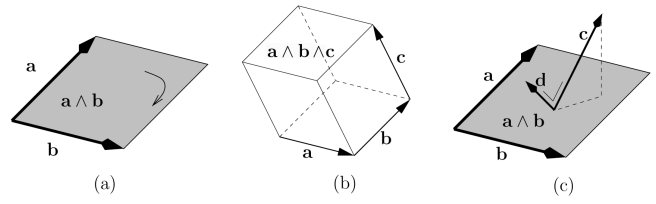


Figure 2. (a) The outer product of two vectors \mathbf{a} and \mathbf{b} results in a 2-blade. The scale of the blade is equal to the size of parallelogram spanned by \mathbf{a} and \mathbf{b} . Although depicted as a parallelogram, $\mathbf{a} \wedge \mathbf{b}$ has no specific shape, only orientation and scale. (b) The outer product of three vectors results in a 3-blade. The scale of the blade is the volume of parallelepiped spanned by \mathbf{a} , \mathbf{b} and \mathbf{c} . (c) The inner product $\mathbf{d} = \mathbf{c} \cdot (\mathbf{a} \wedge \mathbf{b})$ removes everything that is ‘like \mathbf{c} ’ from $\mathbf{a} \wedge \mathbf{b}$, hence the result \mathbf{d} is orthogonal to \mathbf{c} and lies in the $\mathbf{a} \wedge \mathbf{b}$ -plane.

The *dual* of a blade \mathbf{B} is its orthogonal complement and it can be defined as:

$$\mathbf{B}^* = \mathbf{B} \cdot \mathbf{I}^{-1},$$

where \mathbf{I}^{-1} is the inverse of the *pseudoscalar*. The pseudoscalar \mathbf{I} is the unit top-grade blade of the algebra.

For vectors, the *geometric product* (denoted by a half space) is the sum of the inner and outer products:

$$\mathbf{a} \mathbf{b} = \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \wedge \mathbf{b}.$$

Note that a versor is not of a single grade: the versor above has a grade 0 part ($\mathbf{a} \cdot \mathbf{b}$) and a grade 2 part ($\mathbf{a} \wedge \mathbf{b}$). This is similar to the real and imaginary part of a complex number, and the versor above indeed behaves very much like a complex number.

Although we introduce the geometric product as the sum of the inner and outer products, it is actually the most fundamental product of geometric algebra from which most other products are derived by *grade selection*.

Versors are used to represent and apply transformations. Examples from the conformal model are rotations and translations. Invertible blades are versors too, and they can also be used to perform transformations. For example, a conformal plane can be used to perform a reflection in that plane. Because of the nice properties of versors, ideally we would like to represent all transformations that define a geometry (in the ‘Kleinian’ sense) by versors. Unfortunately this is not yet possible for every type of geometry.

To get an idea of how versors can represent transformations, note how the geometric product can be used to perform reflections in (unit) vectors:

$$\begin{aligned} \mathbf{a} \mathbf{b} \mathbf{a} &= \mathbf{a} (\mathbf{b} \cdot \mathbf{a} + \mathbf{b} \wedge \mathbf{a}) = \\ \mathbf{a} \cdot \mathbf{b} \mathbf{a} + \mathbf{a} \cdot \mathbf{b} \mathbf{a} - \mathbf{a} \cdot \mathbf{a} \mathbf{b} &= 2\mathbf{a} \cdot \mathbf{b} \mathbf{a} - \mathbf{a} \cdot \mathbf{a} \mathbf{b}. \end{aligned}$$

You may recognize the RHS of this equation as the classic reflection equation. So, the ‘sandwiching’ of \mathbf{b} between \mathbf{a} results in the reflection of \mathbf{b} in \mathbf{a} . Sandwiching is the typical way of applying a versor to another algebraic element. To make this work for non-unit vectors, we introduce *inversion*:

$$\mathbf{A}^{-1} = \widetilde{\mathbf{A}} / (\mathbf{A} \widetilde{\mathbf{A}}),$$

such that:

$$\mathbf{A}^{-1} \mathbf{A} = \mathbf{A} \mathbf{A}^{-1} = (\mathbf{A} \widetilde{\mathbf{A}}) / (\mathbf{A} \widetilde{\mathbf{A}}) = 1.$$

Since blades can always be re-factored as the geometric product of vectors, applying a versor to a blade yields:

$$\mathbf{a} (\mathbf{b} \wedge \mathbf{c}) \mathbf{a}^{-1} = \mathbf{a} \mathbf{b} \mathbf{c} \mathbf{a}^{-1} = \mathbf{a} \mathbf{b} \mathbf{a}^{-1} \mathbf{c} \mathbf{a}^{-1}.$$

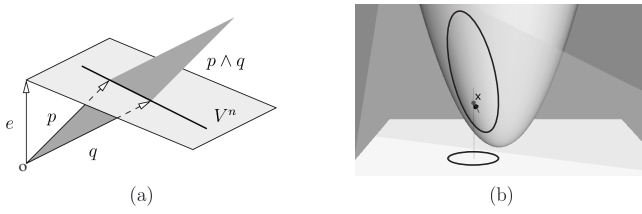


Figure 3. (a) Representing flat subspaces that do not contain the origin in the homogeneous model. Here, the outer product of two points p and q represents a line by its intersection with the ‘Euclidean plane’ that is offset a distance e from the origin. (b) Representing round subspaces in the conformal model. The dual of vector x represents a plane. The intersection of the plane with the paraboloid is down-projected to form a circle in the ‘Euclidean plane’ at the bottom of the figure. This circle is how we interpret x .

The RHS of the last equation shows that applying some transformation to a blade yields the same result as applying the transformation to the factors of the blade. This is a very important concept in GA.

Rotations can be constructed as the concatenation of two reflections, so any blade \mathbf{X} can be rotated as follows:

$$\mathbf{a} (\mathbf{b} \mathbf{X} \mathbf{b}^{-1}) \mathbf{a}^{-1} = (\mathbf{a} \mathbf{b}) \mathbf{X} (\mathbf{b}^{-1} \mathbf{a}^{-1}) = \mathbf{R} \mathbf{X} \mathbf{R}^{-1}.$$

You may recognize \mathbf{R} as a quaternion, but these elements are called *rotors* in geometric algebra. A rotor is a type of versor.

4. The Conformal Model

The conformal model is an $N + 2$ dimensional model of N dimensional conformal geometry. So 3D geometry would be embedded in a 5D algebra. Even though geometric algebra is independent of dimension, we use the 3D case as an example throughout this paper. The conformal model is an extension of homogenous coordinates. It gets its power mainly from two techniques. The first is the way points are represented: they lie on a paraboloid in one of the extra dimensions. The second technique is using a special metric that allows versors to represent much more than ordinary rotations.

To represent subspaces that do not contain the origin, the conformal model uses one of the extra dimensions. The extra dimension allows points, lines and planes to be represented as 1-, 2- and 3-blades respectively (or, without GA, they can be represented as ‘Plücker coordinates’). Even though homogeneous blades still contain the origin of the algebra, we *interpret* them as their intersection with a plane at a unit distance from the origin. See Figure 3a.

In the conformal model, the second extra dimension is used for a similar trick. Only this time we do not interpret blades by their intersection with a plane, but by the down-projection of their intersection with a 3D paraboloid (called the *horosphere*). This allows us to represent not only ‘flat’ objects like lines and planes, but also ‘round’ objects such as circles, spheres and point pairs. See Figure 3b.

In conformal GA, the extra basis vector for the first extra dimension is called o , because it represents the point at the origin. The second extra dimension is denoted by ∞ . ∞ is translationally invariant and it represents the point at infinity. In code, we denote o and ∞ with `no` and `ni`, respectively. This stands for Null vector Origin and Null vector Infinity.

Points lie on the paraboloid, so a point at Euclidean location \vec{p} is represented by a vector:

$$\mathbf{p} = o + \vec{p} + \frac{1}{2}\vec{p}^2\infty.$$

From this it is clear that as \vec{p} approaches infinity, $\frac{1}{2}\vec{p}^2\infty$ will become the dominating term of the point.

Once we have points, we can immediately construct lines, planes, point pairs, circles and spheres by computing the outer product of the minimal number of points that define the surface of these objects. For example, a circle is constructed as the outer product of three points:

$$\mathbf{C} = \mathbf{a} \wedge \mathbf{b} \wedge \mathbf{c}.$$

For flat objects, the point at infinity must be included. E.g., a line goes through 2 points and infinity:

$$\mathbf{L} = \mathbf{a} \wedge \mathbf{b} \wedge \infty.$$

The conformal model uses a special *metric* for basis vectors o and ∞ :

$$o \cdot o = 0, \quad \infty \cdot \infty = 0, \quad o \cdot \infty = -1.$$

This metric is such that the inner product of two points is proportional to their Euclidean distance squared. Let \vec{p} and \vec{q} be two regular Euclidean vectors that point to two locations. Then the inner products of two points at those locations is:

$$\begin{aligned} (o + \vec{p} + \frac{1}{2}\vec{p}^2\infty) \cdot (o + \vec{q} + \frac{1}{2}\vec{q}^2\infty) &= \\ -\frac{1}{2}\vec{p}^2 + \vec{p} \cdot \vec{q} - \frac{1}{2}\vec{q}^2 &= -\frac{1}{2}(\vec{p} - \vec{q})^2. \end{aligned}$$

The RHS of this equation is the distance between the points $(\vec{p} - \vec{q})$, squared, and multiplied by a factor $-\frac{1}{2}$. This technique was already known as a trick to speed up point-distance computations, but is thoroughly embedded in the conformal model.

5. Numerical GA Implementation

In this paper we are concerned only with numerical GA implementation, and limit ourselves to low dimensional geometric algebras ($N < 10$): the method we use here is ineffective for high dimensional algebras because it uses 2^N coordinates to represent a multivector. The method is the most common one used, but other implementation methods exist. Some of those are basically equivalent to the one described here, while others are aimed at specifically at high-dimensional geometric algebras.

We consider the following to be the minimum requirements for any general-purpose GA implementation:

- representation of multivectors.
- implementation of the following operations:
 - negation, reversion, Clifford conjugate and grade involution. These are unary operations that (selectively) toggle the sign of coordinates.
 - inversion
 - addition
 - geometric product, and its derived products:
 - outer product,
 - various inner products
 - meet and join of blades.
 - exponential

Together the above operations will be referred to as ‘the operations’ of the algebra.

The non-linear *meet* and *join* compute the union and intersection of blades. The exponential is used for computing transformations. We do not discuss these operations in detail here.

The rest of this section gives a short overview of implementation internals. The goal is to show the essence of such an implementation: the optimizations that Gaigen 2 applies can be understood without knowing the precise details.

5.1 The Levels of the Implementation

Multivectors are represented as a weighted sum of orthogonal basis blades. Many GA operations are distributive, so we can trivially implement them if we first implement them for basis blades. The non-linear and non-distributive operations require special algorithms that are built on top of the distributive operations. Accordingly, the problem of GA implementation can be divided into four ‘levels’:

- 0 Representing multivectors.
- 1 Implementing distributive operations for basis blades.
- 2 Implementing the distributive operations for multivectors.
- 3 Implementing the non-distributive operations for multivectors.

We do not discuss level 3 here because it is above the level at which Gaigen 2 applies its optimizations.

5.1.1 Level 0: Multivector Representation

2^N orthogonal basis blades are required to span a basis for an N dimensional GA. An example of a basis for a 3D GA is:

$$\left\{ \underbrace{1}_{\text{grade 0}}, \underbrace{e_1, e_2, e_3}_{\text{grade 1}}, \underbrace{e_1 \wedge e_2, e_2 \wedge e_3, e_3 \wedge e_1}_{\text{grade 2}}, \underbrace{e_1 \wedge e_2 \wedge e_3}_{\text{grade 3}} \right\}.$$

Here e_1 , e_2 and e_3 are the basis vectors in the x , y and z directions. Given such a basis, multivectors can be represented as a list of coordinates that refer to a list of basis blades. We can formalize this by defining a row matrix $[\mathbf{L}]$

$$[\mathbf{L}] = [1, e_1, e_2, e_3, e_1 \wedge e_2, e_2 \wedge e_3, e_3 \wedge e_1, e_1 \wedge e_2 \wedge e_3].$$

A multivector \mathbf{A} is then represented by a column matrix $[\mathbf{A}]$ which contains the coordinates:

$$\mathbf{A} = [\mathbf{L}] [\mathbf{A}].$$

2^N coordinates per multivector variable may seem like a lot, but the structure of geometric algebra is such that multivectors are quite sparse. It is possible to significantly ‘compress’ the coordinates, and two compression methods are in use:

1. Per-coordinate compression: a coordinate is only stored and processed when non-zero. This compression method is most efficient in terms of storage, but it is time-consuming during (de-)compression and other operations, because each coordinate must be handled individually.
2. Per-grade compression: the coordinates for a grade part are not stored when all coordinates for that grade part are 0. Although this technique may store some coordinates that are zero, it is often more time-efficient because coordinates can be processed in relatively large blocks (i.e., per grade), instead of one-by-one.

5.1.2 Level 1: Operations on Basis Blades

To implement the distributive operations on basis blades, a representation of basis blades is a first requirement. For this purpose the *bitmap representation* is often used (Figure 4). Each bit in the bitmap signifies the presence of a specific basis vector. If e_i is present as a factor in the basis blade, then bit 0 in bitmap is true. If e_j is present as a factor in the basis blade, then bit $(j - 1)$ in bitmap is true. As many bits are required as the dimension of the algebra. The bitmap itself can not represent scale, so a separate floating point number is required to represent a *weighted* basis blade.

With the basis blades represented as bitmaps, the distributive operations are implemented as low-level bit manipulation functions. For example:

- The geometric product of basis blades is a bitwise exclusive or (XOR) in the bitmap representation.

basis blade	bitmap representation
1	0_b
e_1	1_b
e_2	10_b
$e_1 \wedge e_2$	11_b
e_3	100_b
$e_1 \wedge e_3$	101_b
$e_2 \wedge e_3$	110_b
$e_1 \wedge e_2 \wedge e_3$	111_b
e_4	1000_b
...	...

Figure 4. The bitmap representation of basis blades. Note how well the representation scales up with the dimension of the algebra.

- Operations like reverse selectively flip the sign of the weight, based on the number of 1-bits in the bitmap.

It may seem computationally expensive to use this as the basis of the implementation. However, most implementation ‘hide’ some or all of this level in one way or another: it is hard coded into tables or pre-computed at run-time when the implementation initializes. Our Gaigen 2 performs all level 1 computations at code-generation time, and encodes the result directly into generated source-code.

5.1.3 Level 2: Distributive Operations

We stated that most GA operations are distributive. As an example, take the outer product and reversion:

$$(\mathbf{a}_1 + \dots + \mathbf{a}_n) \wedge (\mathbf{b}_1 + \dots + \mathbf{b}_m) = \sum_{i=1}^n \sum_{j=1}^m \mathbf{a}_i \wedge \mathbf{b}_j,$$

$$\text{reverse}(\mathbf{b}_1 + \dots + \mathbf{b}_n) = \sum_1^n \text{reverse}(\mathbf{b}_i).$$

Because we store multivectors as the weighted sum of basis blades, we can literally distribute the actual work down to the basis blade level.

6. Problems with Efficient GA Implementation

There are a number of issues inherent to geometric algebra that make it hard to create an efficient and easily usable implementation. We shall first give an inventory of the problems and then discuss existing implementation efforts.

6.1 Complicating Factors

- Directly using multivectors as elements of computation is too general. A multivector can represent any multivector type in a geometric algebra. This abstraction is great from a mathematical point of view, but it enforces overhead on the representation (i.e., compression).
- The number of basic operations on multivectors is quite large. Preferably we would like every operation to work on every multivector type, but this leads to a combinatorial explosion when encoded individually for maximum efficiency.
- The metric can be non-Euclidean. Non-Euclidean metrics are essential for many GA-based models. Looking up the metric at run-time from a table is too costly, so preferably we want the metric to be coded directly into the implementation.

There are also some issues that affect all algebras with ‘small’ elements:

- The execution of each individual product or operation requires relatively few computations. Therefore any overhead imposed

by the implementation (such as a conditional branch due to looping) results in a significant degradation of performance.

- Often, expressions consisting of multiple products and operations can be executed much more efficiently by folding them into one calculation rather than executing them one by one through a series of function calls.

6.2 Existing Work

Quite a few geometric algebra implementations exist. We list a number of them here, focussing on their performance. All these implementations are variations of the method described in Section 5.

- **Gaigen 1** is our previous GA code generator. It generates C++ implementations of geometric algebras, where the dimension of the algebra is limited to 8. Beyond 8D, the generated code gets so large that most compilers have problems compiling it. The generated code can be optimized for specific applications through profiling. Gaigen 1 uses per-grade compression for multivectors. As a result, Gaigen 1 is $2\times$ to $5\times$ slower than traditional geometry implementations [4].
- **GABLE** (Geometric Algebra Learning Environment) is a Matlab library intended purely for educational purposes. GABLE does not use coordinate compression, and in order to evaluate products, it constructs up to $2^N \times 2^N$ matrices from the multivector coordinates. Hence GABLE is not very efficient: in one application [5] which was initially developed in GABLE we experienced a $6000\times$ speed up by porting to Gaigen 1.
- **CLU** is an extensive GA library written in C++, with visualization and an interactive visual calculator. It is one of the most feature-complete GA libraries. It uses per-coordinate compression. In [4] we showed that it was about $30\times$ slower than Gaigen 1.
- **MV** [6] is a C++ library which uses per-coordinate compression. It works up to very high dimensions ($N \leq 64$), although the feasibility of using such high-dimensional algebras depends on how well the problem is ‘aligned’ with the basis of the algebra. We used a synthetic benchmark to determine MVs speed relative to Gaigen 1. We found that for very low dimensional algebras ($N \leq 5$), the performance of Gaigen 1 and MV is about equal. As the dimension increases, Gaigen 1 becomes more and more efficient relative to MV, up to $20\times$ faster for $N = 8$. MV is likely the fastest library that uses per-coordinate compression, and the only library that handles both low- and high-dimensional algebras well.
- **boost::math::clifford** is a proof-of-concept library which implements geometric algebra using C++ meta-programming. It is based on some of the same ideas as Gaigen 2 (specialization of multivectors, instantiating and optimizing functions). Together with the author of the library we performed synthetic benchmarks on 3D geometric algebras. We determined that Gaigen 2 and Clifford basically have the same performance. However, Clifford does not allow for non-orthogonal basis vectors, which puts it at a disadvantage for use with the Conformal model (see the benchmark in Section 8.4).

Except for `boost::math::clifford` and Gaigen 1, these libraries all suffer from the problems listed in the previous section. `boost::math::clifford` has none of the problems. Gaigen 1 does encode the metric directly into the implementation, and tries to optimize functions by adapting them to the context of the full application. However, it only offers the multivector as element of computation, which slows down execution due to the compression.

6.2.1 Related Code Generators

There are several code generators that produce implementations of linear algebra (e.g. ATLAS[7]) and signal transforms (i.e., SPIRAL [8]). Like Gaigen 2, SPIRAL uses a DSL to describe functions. These generators focus on optimization for specific hardware (i.e., they take into account the behavior of the cache of the CPU). Gaigen 2 produces generic (C++) code and leaves hardware optimization to the compiler. The optimizations that these generators apply could possibly be applied to Gaigen 2 to improve performance.

7. Efficient GA Implementation in Gaigen 2

Gaigen 2 is our second Geometric Algebra Implementation Generator. It is a stand-alone code generator that generates GA implementations (source code) from high level specifications. While designing Gaigen 2, we found performance especially important: we wanted to prove that geometry implemented through GA does not have to be significantly slower than geometry implemented through more traditional methods. At the time Gaigen 1 was developed, it the fastest GA implementation available, yet still $2\times$ to $5\times$ slower than traditional methods. Our intent was to reduce that to less than $1.5\times$. To overcome the performance problems of Gaigen 1, we had two straightforward goals in mind:

- Minimize the number of coordinates used to store multivector variables.
- Avoid all conditional statements in the optimized parts of the generated code.

Minimizing the number of coordinates can be done through compression, but this causes a lot of conditional statements which slow down execution. Our solution is to generate a specialized multivector class for each multivector type that is in use in the program. A specialized multivector class is often much more efficient (both in storage requirements and processing) than a general multivector. E.g., to store a 3D line in the conformal model, only 6 coordinates are required, where a general non-compressed multivector would require 32 coordinates (of which at least 26 would be zero).

Once specialized classes are available, multivector variables should be statically typed by the user. So instead of writing (in for example C++):

```
multivector v; // a vector
multivector C; // a circle
```

the user writes:

```
vector v;
circle C;
```

Functions over general multivectors can then be instantiated with the specialized classes. This way, we can have the benefit of coordinate compression without any conditional statements. Because there are no conditionals due to compression, the compiler also has better opportunities to optimize.

Gaigen 2 takes as input:

- a specification of a geometric algebra (metric, dimension, etc),
- a set of functions over multivectors,
- and optionally a profile that contains information about what functions to optimize for specific types of arguments.

Gaigen 2 outputs a set of source files in the desired language. The languages are currently limited to C++ or Java, but a new back-end can be written in about a week.

Because new types had to be created from specification, and functions had to be instantiated with these types, some form of generative programming was required. Although C++ meta-

programming has been successfully used in a number of numerical libraries (e.g., Blitz++ [9], FTensor, MTL and Clifford), our desire to obtain a cross-language implementation ruled out language-specific techniques. There were also other reasons why we decided to go for a stand-alone code generator:

- More control over the code generation process, which makes it easier to add optimizations.
- The results of the code generation process are visible in the form of easily readable source code.
- More control over final compile process: what to inline, what not to inline, etc. In our experience, C++ meta-programming compiles very slowly, and we wanted to avoid this.
- The possibility to use of DSL to encode the functions.

Gaigen 2 can not perform static analysis of the full application (i.e., in order optimally adapt the generated code without actually running the program), because this would require us to include a parser / analyzer for every target language. So in order to adapt the generated code to the rest of the application, profiling is required: the application is run with a representative input, and Gaigen 2 records what optimizations it needs to do. This profiling step is the biggest drawback of using a stand-alone code generator.

7.1 Example

Before describing Gaigen 2 in detail, let us first do a simple example where we step through every step of the code generation process that Gaigen 2 performs.

Suppose the user defines a specialized multivector type *vector* :

```
// in the algebra specification (written by user):
specialization: vector(e1, e2, e3);
```

This causes a class to be generated that can contain the 3 coordinates of the vector:

```
// generated C++ code:
class vector {
    // ...
    float m_c[3]; // holds e1, e2, e3 (x, y, z) coordinates
};
```

The user also defines a function `add()` in the DSL of Gaigen 2:

```
// in the function definitions (in DSL, written by user):
mv add(mv a, mv b) {
    // the '+' operator is a built-in operator of the DSL
    return a + b;
}
```

This triggers Gaigen 2 to emit a function:

```
// generated C++ code:
mv add(const mv &a, mv &b) {
    // Inefficient code which adds 'a' and 'b'
    // (lots of conditional statements)
    // ...
}
```

This function is very inefficient because works with general multivectors. It contains many conditional statements (not shown here) that deal with compression of general multivectors 'a' and 'b'. If the user now writes:

```
// C++ code (written by user):
vector a, b;
// ...
vector c = add(a, b);
```

then the C++ compiler will select the inefficient `add(const mv&, const mv&)` function from above as 'best viable'. The compiler will do this because the `mv` class has a converting constructor from `vector` to `mv`, and no other function is viable. So the application will work, but not very efficiently.

To improve efficiency, a specialized `add()` function for `vectors` should be generated. The user can instruct Gaigen 2 to do so by adding a line to the profile:

```
// in profile:
usage : mv add(mv a, mv b) : vector, vector;
```

The part of the line before the colon identifies the function, the rest of the line identifies the specialized argument types. Gaigen 2 will take the DSL function `mv add(mv a, mv b)`, replace its argument types with `vectors` and optimize it. The end-result is the following efficient C++ function:

```
inline vector add(const vector &a, const vector &b) {
    return vector(
        a.c[0] + b.c[0],
        a.c[1] + b.c[1],
        a.c[2] + b.c[2]);
}
```

This function can be so efficient and simple because it deals with statically typed `vectors` instead of general multivectors, avoiding compression issues.

7.2 Detailed Description

7.2.1 Algebra Specification

The specification of the algebra consists of:

1. The basis of the algebra. This lists basis vectors of the algebra and defines their metric. For example, for a conformal geometric algebra:

```
basis: no, e1, e2, e3, ni;

metric: e1 . e1 = e2 . e2 = e3 . e3 = 1;
metric: no . ni = -1;
```

If inner products between basis vectors are not specified, they are assumed to be 0. `no` and `ni` are the basis vectors for the origin and infinity. Contrary to what is written in Section 5.1.2, Gaigen 2 allows the use of a *non-orthogonal* basis. This complicates the job of the code generator, but can improve run-time performance (see Section 8.4).

2. Definitions of specialized multivectors. A specialized multivector type is defined by the name of the specialization plus a list of basis elements whose respective coordinates can be non-zero. For example:

```
specialization: plane(e1^e2^e3^ni, e1^e2^no^ni,
    e1^e3^no^ni, e2^e3^no^ni);
```

This line defines a specialized multivector type which can hold a conformal plane. A plane has four coordinates which refer to basis blades $e_1 \wedge e_2 \wedge e_3 \wedge \infty$, $e_1 \wedge e_2 \wedge o \wedge \infty$, etc.

3. What type of instrumentation to include in the generated code. For example, profiling instrumentation and 'visual debugging instrumentation' are currently available. Visual debugging instrumentation enables visual rendering of the geometry inside a (running) application.
4. Options that depend on the target language, such as what part(s) of the source code to inline, what namespace or package to put the generated code in, how to bind operators, what type of floating point numbers (`float`, or `double`) to use for the coordinates, etc. For example, in C++ the user could choose to inline everything for high performance, or to inline nothing for faster compile times.

7.2.2 General Multivector Implementation

In high-performance applications, the general multivector class should not be used. Still it is a very important part of the generated

code. As long as the code is not (fully) adapted to the rest of the application, specialized multivectors are converted to general multivector in order to evaluate non-optimized functions. Also, functions that we do not consider to be performance-critical (such as pretty printing) are generated only for general multivectors.

The coordinates of a general multivector are stored using a per-grade compression scheme. A bitmap is used to keep track of what grade parts are present.

When profiling instrumentation is to be included, the class is generated with an extra integer field that holds the specialized multivector type of the multivector. Each specialized multivector is identified by a unique integer. When a specialized multivector variable is converted to a general multivector variable, the general multivector variable will carry this integer in its `type` field. This information is used for profiling, as described in Section 7.2.7.

A function is provided to compress a (non-compressed) list of coordinates, i.e., to initialize a multivector from its 2^N coordinates. This function is used when a general multivector is initialized from user-supplied coordinates. It is also used internally in non-optimized functions, to compress the result of computations.

Multivector variables can be pretty printed to human readable form, e.g. $0.81 - 2.4 * e1 \wedge e2 + 1.3 * e2 \wedge e3$. This output can also be parsed back in again, using an ANTLR parser. The grammar for this parser is generated by Gaigen 2 along with the rest of the algebra implementation source code.

7.2.3 Specialized Multivectors

The specialized multivector classes are straightforward: they only serve as a container for their non-zero, non-constant coordinates. An example of such a class (`vector`) was already shown above.

Constants coordinates for specialized multivector types have not yet been introduced: The definition of a specialized multivector type may state that certain coordinates have a constant value, e.g.:

```
// in algebra specification:
specialization: normalizedPoint(no = 1, e1, e2, e3, ni);
```

The value of the `no` coordinate of a *normalized point* is always '1' so there is no need to store this value in every `normalizedPoint` variable. When a coordinate is defined constant, Gaigen 2 does not reserve storage in the specialized multivector class to store it. Instead, it encodes the constants directly into optimized DSL functions. This saves both storage and processing time.

Specialized multivectors can also be fully constant, in which case they serve only as a 'symbolic token'.

7.2.4 Function Definition in DSL

A domain specific language is used to define the functions over multivectors. The language is a C-like language with built-in GA-functions and GA-operators. A special expression parser allows for re-using the standard C operators as GA operators, such that they both work an intuitive way. The rest of the language is straightforward. The language has loops, switches, and if-else statements. All of these are subject to static evaluation during optimization.

7.2.5 Code Generation of Functions

The functions defined in the DSL are to be transformed into functions in the target language. Gaigen 2 always generates a generic (but slow) version that works on all multivector types. Optionally several optimized ones that work on specialized multivectors.

A function is transformed as follows:

1. As a first step, the function can be instantiated with specialized multivector types, as specified in the profile.
2. At this point, high-level coordinate-free optimizations can be applied, but we have not yet implemented any such optimizations.

3. The multivector variables in the function are 'written out' on the basis. This means that each variable is replaced by its value on the basis. For example, a vector variable would be replaced by $c1 * e1 + c2 * e2 + c3 * e3$, where the c_i are symbolic tokens the coordinates of the vector.
4. Low-level optimizations are performed. A rewrite system simplifies the expressions and statically evaluates all (GA) operations on the basis blades as far as possible. This is where 'level 1' (Section 5.1.2) of the GA implementation is executed. The rewrite system also unrolls loops, and statically evaluates `if else` and `switch` statements, whenever possible.

When the low-level rewriting is finished, all GA-operations have been removed: only standard operations (multiply, add, divide, trigonometry) on regular types (floats, integers, booleans) remain. The code is now ready for conversion to the target language.

7.2.6 Emitting Code in the Target Language

The functions are handed over to a custom back-end that emits the code in the target language. Emitting the code of the optimized DSL functions amounts to converting their abstract syntax tree to the equivalent AST in the target language. The resulting AST is then pretty printed.

The back-end also generates all the 'boiler plate' code that defines all multivectors classes and basic other code. This is done mostly from active templates. These templates are a mix of the actual code that should be emitted, and small Java programs which control the behavior of the template – similar to JavaServer Pages.

7.2.7 Profiling

For optimal performance, the generated source code must be adapted to the application which links to it. DSL functions over multivectors must be instantiated with specific multivectors types and optimized. Because Gaigen 2 can not perform a static analysis of the application source code, it uses a profiling to extract the required information from the application. Profiling an application proceeds as follows:

- The source code is (re-)generated with extra profiling instrumentation.
- The application is recompiled and run through a representative input. The instrumentation records the usage profile of DSL functions: the specialized multivector types of the arguments, and the number of invocations.
- The profile is stored in a file, and Gaigen 2 regenerates the source code, this time without profiling instrumentation, but with all required optimizations.
- The application is recompiled.

Although this may seem a straightforward solution, there is a problem that prevents it from working well. If the application uses nested calls to DSL functions, the profiling information can not be determined in a single run of the program. When DSL functions are instantiated and optimized, the return type changes from general multivectors to some specialized multivector type. Gaigen 2 determines this specialized type by analyzing all possible return values of the function. Note that the generated code can not easily determine the return types at run-time, because this is a non-trivial computation.

Thus the profiling process described above must be repeated multiple times until the type information has 'bubbled' through all nested function calls. This is illustrated in Figure 5. It shows the abstract syntax tree for the function calls in the following piece of code.

```
// C++ code, written by user
```

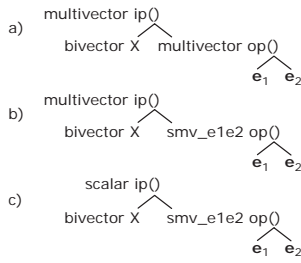


Figure 5. C++ AST of example code during various stages of optimization.

```
bivector X;
ip(X, op(e1, e2));
```

`ip()` and `op()` are both DSL functions. When the generated code is not optimized (i.e., no profile information), the C++ AST looks like Figure 5a. After the first profiling run, an optimized function for the call `op(e1, e2)` has been created. It returns a newly generated type `smv_e1e2`. Gaigen 2 makes up symbolic names for the types it has to create on the fly. In this example, it determines that the outer product of constant `e1` and constant `e2` is another constant `e1 \wedge e2`. The code is regenerated, the application recompiled and linked again, and in the second profiling run, the use of function `ip(bivector, smv_e1e2)` is recorded (Figure 5b). This function is instantiated (return type `scalar`) and after another round of code generation and compilation the final result is Figure 5c.

In our own applications, we have found that nested function calls can easily be five levels deep, thus requiring five iterations of the profiling process. We found this very tiresome, and to avoid it we have adjusted the profiling instrumentation as follows.

When the application starts, the profiling instrumentation connects to Gaigen 2, which is running in the background. On each DSL function invocation, the profiling instrumentation sends the ID of the functions and the type-IDs of the arguments to the code generator. The code generator instantiates the function with the arguments, and determines the specialized multivector return type. The ID of the return type is sent back to the profiling instrumentation, which uses it to set the type field of the multivector variable which is returned by the function. This way, general multivectors will always have a specialized multivector type (unless of course the user explicitly creates a general multivector without any type).

When the application terminates, the profiling connection to Gaigen 2 is closed. In response, Gaigen 2 stores the profile in a file, and optionally regenerates the algebra implementation immediately.

7.3 Code Generation Process Overview

The full Gaigen 2 code generation process is illustrated in Figure 6. In the top left corner are the algebra specification and the DSL function definitions. These are processed by the various stages of Gaigen 2 (the big block in the middle). The resulting source files (on the right) are linked to the rest of the application. If required, the application is profiled: the profiling information and type IDs are sent back and forth over the network connection between the application and Gaigen 2. The generated code is regenerated with the correct optimizations, after which the application is compiled and linked again.

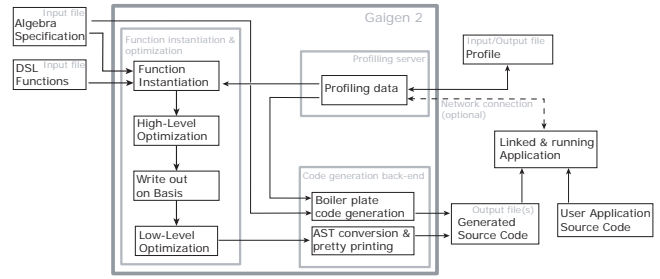


Figure 6. The full Gaigen 2 code generation process.

8. Results

We have performed a large number of benchmarks to measure the performance of Gaigen 2 relative to other ways of implementing geometry.

8.1 Ray Tracer

The following figures were obtained by benchmarking a full application, not through synthetic benchmarks. A basic ray tracer was implemented multiple times, each time using a different model or implementation of geometry. This allowed for comparison of various methods (LA vs GA), models (3D, 4D homogeneous, 5D conformal) and geometry implementations (standard¹, Gaigen, CLU, Gaigen 2) in a realistic application. Compared to the original paper [4] on Gaigen 1 (the lower part of the table), the most notable improvement is a reduction of the cost of using the conformal model to just 25%.

model	implementation	rendering time relative to 3DLA
3D LA	standard	1.00×
4D LA	standard	1.22×
3D GA	Gaigen 2	0.98×
4D GA	Gaigen 2	1.2×
5D GA	Gaigen 2	1.26×
3D GA	Gaigen 1	2.56×
4D GA	Gaigen 1	2.97×
5D GA	Gaigen 1	5.71×
3D GA	CLU	78×
5D GA	CLU	178×

8.2 Inverse Kinematics

GA is not always slower than LA. We implemented inverse kinematics algorithm [10] using Gaigen 2. The algorithm is based on conformal geometric algebra. It was designed as a replacement of an existing LA-based algorithm [11]. Our result was 43% faster than the existing solution. This improvement is mostly due to the fact that GA has a superior representation for rotations built right into the algebra. The LA implementation had to convert back and forth between matrices and quaternions.

8.3 Singularity Detection

[5] describes an algorithm for detecting singularities in vector fields. For this purpose, 3D GA is used. The original implementation was based on GABLE, and was extremely slow. We were asked to port the algorithm to Gaigen 1 and the port performed approximately 6000× faster than the original. A subsequent port from Gaigen 1 to Gaigen 2 tripled the performance.

¹ ‘standard’ means that the linear algebra implementation was hand written, inlined, optimized C++ code.

8.4 Choice of Basis Vectors

To show the importance of picking the right basis vectors we present one more (synthetic) benchmark.

In the conformal model, the vectors o and ∞ are very important, as they represent the origin and infinity. However, o and ∞ are not orthogonal (i.e., $o \cdot \infty = -1$). Hence, using o and ∞ as basis vectors complicates the computation of products of basis blades (Section 5.1.2). Thus, most GA implementations use a different basis: Two orthogonal vectors e_+ and e_- are used. Their metric is defined by $e_+ \cdot e_+ = 1$, $e_- \cdot e_- = -1$. o and ∞ are then constructed as a weighted sum of e_+ and e_- .

The problem with this approach is that the structure of the conformal model makes it more efficient to use the $o\infty$ -basis. For example, the flat objects (line and planes) contain ∞ as a factor. This means that they have fewer coordinates on the $o\infty$ -basis than on the e_+e_- -basis.

We have written a synthetic benchmark which demonstrates the effect of the choice of basis. Our hypothesis was that translating lines would be affected by the choice of basis vectors because both the lines and the ‘translators’ have ∞ as a factor. Rotating circles should not be affected, as neither rotors nor circles have ∞ as a factor. The example was designed specifically for this purpose, so the effect is exaggerated:

	$o\infty$ -basis	e_+e_- -basis
translating lines	0.26s	0.81s
rotating circles	0.45s	0.46s

The times are for repeating the operation 3 million times. Translating lines is more than $3\times$ more efficient on the $o\infty$ -basis than on the e_+e_- -basis. As expected, the choice of basis vectors does not significantly affect the performance of rotating circles.

9. Discussion

We used a stand-alone code generator instead of an existing framework such as C++ meta-programming. The downside (for the implementor) is that programming a stand-alone code generator takes significantly more effort than implementing a C++ meta-programming library. However, the end result is much more flexible. We have control over what is inlined. We can easily port to different languages: we implemented a Java back-end in about a week. We can make optimizations that would be very hard in a meta-programming library. For example, using non-orthogonal basis vectors (Section 8.4) requires the computation of the eigenvalue decomposition of a matrix.

We expect that the Gaigen 2 framework can easily be applied to other problem domains such as linear algebra and tensor algebra. This would possibly allow for automatic optimization of complex algorithms such as the SVD and eigenvalue decomposition.

For the end-user of our implementation, the largest drawback of our approach is profiling. We have tried to simplify the profiling process as much as possible: a profile can be obtained in a single run, which requires two recompiles and running the code generator twice. But still, in our own use of Gaigen 2, we find profiling an awkward activity and tend to postpone it as long as possible. We are considering adding a GUI front-end to Gaigen 2 which may simplify the profiling process even more – currently, Gaigen 2 is a command line tool.

Concerning the benchmarks, we are very satisfied with the results. Gaigen 2 is on average $3x$ faster than Gaigen 1, which is more than we expected. Gaigen 2 implementations of the classic geometry models (3D, 4D) are just as fast as their traditional LA counterparts. The conformal model can be somewhat slower in certain applications, but we find this acceptable given the much better semantics.

10. Conclusion

We have shown how a stand-alone code generator can be used to efficiently implement geometric algebra in a language independent way. We synthesize GA implementations from specification and optimize functions described in a DSL, to obtain a very high performance result. The stand-alone nature of the code generator allows us to do optimizations that would be very hard in a C++ meta-programming approach. It also makes Gaigen 2 portable across programming languages.

We have performed extensive benchmarks where we compare Gaigen 2 geometry implementations to their classical (LA-based) counterparts. From those different benchmarks we can conclude that GA-based geometry is 25% slower to 40% faster. The exact results will vary per application: they depend on the problem at hand, the model of geometry that is used, the compiler and other implementation details. In any case, these figures are a significant improvement over previous GA implementations. To the best of our knowledge, Gaigen 2 is currently the fastest general-purpose GA implementation.

To adapt the generated code to the application that links to it, we use a profiling feedback loop that extracts the required information in a single run of the application. The feedback loop is required because new (multivector) types may have to be generated to obtain an optimal implementation. In principle this profiling method is usable for any kind of algebra, but it is especially useful for GA, since it has a non-trivial type system.

Acknowledgments

Many thanks to Dick Grüne for his assistance in writing this paper. Thanks to Dietmar Hildenbrand (inverse kinematics example), Stephen Mann (singularity example), Jaap Suter (Clifford), Ian Bell (MV) and anonymous ‘reviewer 3’ for several useful references.

References

- [1] L. Dorst and D. Fontijne. *An Algebraic Foundation for Object-Oriented Euclidean Geometry*. ITM 2003 proceedings, 2003.
- [2] L. Dorst and S. Mann. *Geometric algebra: a computation framework for geometrical applications: Part I* Computer Graphics and Applications, 2002, Volume 22, May/June 2002, pp. 24-31.
- [3] C. Doran and A. Lasenby. *Geometric Algebra for Physicists*. Cambridge University Press, 2003.
- [4] D. Fontijne and L. Dorst. *Modeling 3D Euclidean Geometry*. IEEE Computer Graphics and Applications, Volume 23, March/April 2003, pp. 68-78.
- [5] S. Mann and A. Rockwood. *Using geometric algebra to compute singularities in 3D vector fields*. IEEE Visualization 2002 Proceedings, 2002, pp 283-289.
- [6] I. Bell. *Ian Bell’s pages on geometric algebra*. <http://www.iancgbell.clara.net/mathsg/geoalg1.htm>
- [7] R. Clint Whaley and Antoine Petitet and Jack J. Dongarra. *Automated Empirical Optimization of Software and the ATLAS Project*. Parallel Computing, 2001, volume 27, issue 1-2, pp 3-35.
- [8] Markus Püschel et al. *SPiRAL: Code Generation for DSP Transforms*. Proceedings of the IEEE, volume 93, issue 2, pp 232-275.
- [9] T. Veldhuizen. *Blitz++: Object-Oriented Scientific Computing*. <http://www.oonumerics.org/blitz>
- [10] D. Hildenbrand and E. Bayro-Corrochano and J. Zamora-Esquivel. *Advanced Geometric Approach for Graphics and Visual Guided Robot Object Manipulation*. Proceedings of ICRA conference, April 2005, Barcelona, Spain.
- [11] D. Tolani and A. Goswami and N. Badler. *Real-time inverse kinematics techniques for anthropomorphic limbs*. University of Pennsylvania, 2000.