Implantation de l'Algèbre Géométrique en Objective Caml

S. Charneau, L. Fuchs & L. Aveneau

SIC, FRE CNRS 1731, Université de Poitiers
SP2MI, Bld Marie et Pierre Curie BP 30179
86962 Futuroscope Chasseneuil Cedex
{charneau, fuchs, aveneau}@sic.univ-poitiers.fr

Résumé

L'algèbre géométrique fournit un formalisme algébrique qui permet une description homogène des algorithmes géométriques. Cependant, la réalisation de bibliothèques logicielles utilisant ce formalisme reste assez délicate.

Dans cet article, nous présentons deux implantations de l'algèbre géométrique en Objective Caml. La première décrit l'algèbre géométrique d'un espace vectoriel prédéfini, tandis que la seconde présente un module générique de l'algèbre géométrique indépendant de tout espace (pseudo) euclidien. Ces implantations sont comparées à une autre écrite en C++, tant au niveau de leur programmation que de leur efficacité.

1. Introduction

L'Algèbre Géométrique, est une ré-interprétation géométrique de l'*Algèbre de Clifford* introduite par William K. Clifford (1845 – 1879). Cette algèbre a ensuite été étudiée, dans les années 1960 [1], dans le but de développer un cadre formel pour faire de la géométrie.

L'algèbre géométrique consiste à faire de la manipulation symbolique pour représenter des relations géométriques. Elle permet notamment de représenter différents systèmes mathématiques utilisés en géométrie comme l'algèbre vectorielle ou les quaternions, dans un unique formalisme algébrique. De même, la spécification des opérations géométriques est indépendante de tout système de coordonnées, ce qui permet de s'abstraire du modèle de calcul et de faciliter la généralisation de ces opérations à toutes dimensions.

Cependant, la richesse des entités manipulées par l'algèbre géométrique rend la réalisation de bibliothèques efficaces délicate [2]. Les implantations de l'algèbre géométrique qui apparaissent aujourd'hui les plus performantes sont celles issues d'un générateur interactif de bibliothèques C++, appelé Gaigen [3]. Cependant, bien que l'inclusion d'optimisations sur la structure permet d'obtenir de meilleurs résultats que d'autre implantations faites dans un langage impératif, celles-ci nécessitent de gérer des structures supplémentaires dont la manipulation engendre des coûts notables.

Cet article présente deux manières d'implanter l'algèbre géométrique dans un langage fonctionnel, pour son utilisation en infographie. La première est spécifique à l'algèbre sur un espace donné ; la seconde est générique, indépendante d'un espace vectoriel.

Dans la première partie, nous abordons de manière intuitive l'algèbre géométrique, pour donner une idée de la structure implantée. Dans la deuxième partie, nous motivons l'utilisation du langage Objective Caml, puis présentons notre implantation de l'algèbre géométrique dans ce langage. Nous mettons alors en relief les avantages de ce langage par rapport au C++ du point de vue de la programmation (clarté du code, facilité de développement, etc.) et du point de vue de l'efficacité, en comparant nos bibliothèques à celles générées par Gaigen dans une application de type lancer de rayons. Ces comparaisons nous permettent de conclure que l'utilisation d'un langage fonctionnel se révèle être plus intéressante.

2. Présentation de l'Algèbre Géométrique

De manière générale, une algèbre est définie à partir d'une structure d'espace vectoriel à laquelle on ajoute une seconde loi interne, dite produit, distributive sur l'addition de l'espace vectoriel. Dans le cas de l'algèbre géométrique, nous manipulons un espace de poly-vecteurs, muni d'une loi d'addition et d'une multiplication par un scalaire, qui déterminent la structure d'espace vectoriel, et du produit géométrique (seconde loi interne).

Un poly-vecteur est une somme de k-vecteurs qui sont une généralisation des vecteurs aux dimensions supérieures à un. En première approche, une analogie peut être faite avec les polynômes vus comme des sommes de monômes. Les k-vecteurs (que l'on appellera parfois *monômes* de l'algèbre) sont définis à partir des vecteurs d'un espace (pseudo) euclidien[1] $\mathcal{E}$ de dimension n. Pour cela nous introduisons le produit externe, noté '$\wedge$'. Ce produit est *associatif* et *distributif* sur l'addition de l'algèbre. Considérons d'abord les vecteurs de $\mathcal{E}$, appelés, dans ce cas, des 1-vecteurs. Le produit externe de deux 1-vecteurs $\mathbf{a}$ et $\mathbf{b}$ est anticommutatif. Si $\mathbf{a}$ et $\mathbf{b}$ sont linéairement dépendants, alors le produit donne le poly-vecteur nul : $\mathbf{a} \wedge \mathbf{b} = 0$. S'ils sont linéairement indépendants, le produit définit un 2-vecteur (ou bivecteur) $\mathbf{A} = \mathbf{a} \wedge \mathbf{b} = -\mathbf{b} \wedge \mathbf{a}$. De façon récursive, un k-vecteur est le produit externe d'un $(k-1)$-vecteur par un 1-vecteur linéairement indépendant des $(k-1)$ 1-vecteurs définissant le $(k-1)$-vecteur, cela jusqu'à la dimension n de $\mathcal{E}$. L'entier k est appelé le *grade* du k-vecteur, le terme *dimension* étant réservé pour parler des espaces vectoriels. Pour k fixé, l'ensemble des k-vecteurs forme un espace vectoriel, noté $\mathcal{G}_n^k$, de dimension $\binom{n}{k}$. En ajoutant à ces espaces l'espace $\mathbb{R} \cong \mathcal{G}_n^0$ des scalaires (qui seront appelés des 0-vecteurs), nous obtenons par somme directe un espace vectoriel $\mathcal{G}_n = \oplus_{i=0}^n \mathcal{G}_n^i$ sur lequel l'algèbre géométrique est définie.

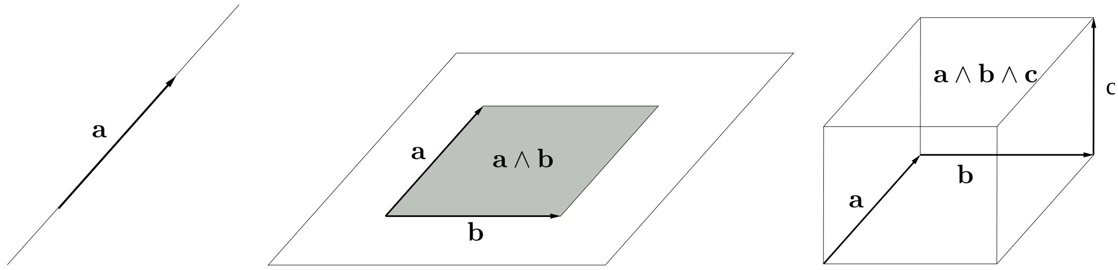
L'interprétation géométrique d'un 1-vecteur est celle d'un segment orienté d'une droite vectorielle. De la même façon, un 2-vecteur $\mathbf{A} = \mathbf{a} \wedge \mathbf{b}$ est interprété comme la portion orientée du plan vectoriel (le parallélogramme) définie par les deux 1-vecteurs $\mathbf{a}$ et $\mathbf{b}$ (voir figure 1). D'une façon générale un k-vecteur s'interprète comme un segment orienté d'un sous-espace vectoriel de dimension k de $\mathcal{E}$. La norme d'un k-vecteur est le volume défini par les k 1-vecteurs dont il est issu.

Si la spécification des opérations géométriques se fait indépendamment d'un système de coordonnées, lors du calcul de l'opération, il est nécessaire d'introduire une base de l'algèbre. Pour construire une base de $\mathcal{G}_n$, nous introduisons une base orthonormée $(\mathbf{e_1}, \mathbf{e_2}, \cdots, \mathbf{e_n})$ de $\mathcal{E}$. Cette base nous permet, à partir du produit externe $\wedge$, de définir une base des k-vecteurs pour chaque espace $\mathcal{G}_n^k$ et, par somme directe des espaces obtenus, de définir une base de $\mathcal{G}_n$. Tout poly-vecteur est alors exprimé dans cette dernière base.

Par exemple, en considérant un espace euclidien $\mathcal{E}$ de dimension 3 et une base orthonormée $(\mathbf{e_1}, \mathbf{e_2}, \mathbf{e_3})$ de celui-ci, nous obtenons dans un premier temps les bases suivantes :

$$\begin{array}{cl} \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3 & \text{pour } \mathcal{G}_3^1 \\ \mathbf{e}_2 \wedge \mathbf{e}_3, \mathbf{e}_3 \wedge \mathbf{e}_1, \mathbf{e}_1 \wedge \mathbf{e}_2, & \text{pour } \mathcal{G}_3^2 \\ \mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3 & \text{pour } \mathcal{G}_3^3 \end{array}$$

[1]Un espace pseudo-euclidien est un espace vectoriel réel, muni d'une forme bilinéaire $\langle\ ,\rangle$. Si la forme quadratique associée à la forme bilinéaire est définie strictement positive, on obtient un espace euclidien muni d'un produit scalaire. La signature de l'espace correspond à la signature de la forme quadratique, il s'agit des dimensions maximales des sous-espaces vectoriels sur lesquels la forme quadratique prend une valeur positive, négative ou nulle.



(a) vecteur, détermine une unique droite

(b) bivecteur, détermine un unique plan

(c) trivecteur, détermine un unique espace tridimensionnel

Figure 1: Interprétation des k-vecteurs dans un espace vectoriel

Ainsi, en ajoutant 1 comme base de $\mathcal{G}_3^0$, nous obtenons la base suivante de $\mathcal{G}_3$:

$$(1, \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3, \mathbf{e}_2 \wedge \mathbf{e}_3, \mathbf{e}_3 \wedge \mathbf{e}_1, \mathbf{e}_1 \wedge \mathbf{e}_2, \mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3)$$

Tout poly-vecteur est ainsi une combinaison linéaire des éléments de cette base.

Pour terminer la construction de l'algèbre géométrique, nous devons définir le produit géométrique. Celui-ci est d'abord défini sur la base $(\mathbf{e_1}, \mathbf{e_2}, \cdots, \mathbf{e_n})$ de $\mathcal{E}$ de la façon suivante :

$$\forall i \in [1;n], \mathbf{e_i}\mathbf{e_i} = \mathbf{e_i} \cdot \mathbf{e_i} = \epsilon|\mathbf{e_i}|^2 \tag{1}$$

$$\forall i,j \in [1;n] \times [1;n], i \neq j, \mathbf{e_i}\mathbf{e_j} = -\mathbf{e_j}\mathbf{e_i} = \mathbf{e_i} \wedge \mathbf{e_j} \tag{2}$$

Dans la propriété (1), $\epsilon \in \{-1, 0, 1\}$ correspond à la signature de $\mathbf{e_i}$ pour la forme quadratique associée à la forme bilinéaire notée '$\cdot$', définie sur $\mathcal{E}$. La propriété (2) nous donne la relation entre le produit géométrique et le produit externe dans une base orthonormée.

À cette définition il faut encore ajouter que le produit géométrique de deux scalaires est le produit de ces deux scalaires, que le produit géométrique d'un scalaire par un vecteur est le produit d'un scalaire par un vecteur et, enfin, que le produit géométrique d'un scalaire par un poly-vecteur est commutatif.

Dans $\mathcal{G}_3$, le produit géométrique de $\mathbf{A} = (\mathbf{e}_1 + 2\mathbf{e}_3 \wedge \mathbf{e}_2)$ et $\mathbf{B} = (-\mathbf{e}_3 + 3\mathbf{e}_1 \wedge \mathbf{e}_2)$ exprimés dans la base $(1, \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3, \mathbf{e}_2 \wedge \mathbf{e}_3, \mathbf{e}_3 \wedge \mathbf{e}_1, \mathbf{e}_1 \wedge \mathbf{e}_2, \mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3)$ est alors :

$$\begin{aligned}
\mathbf{AB} &= (\mathbf{e}_1 + 2\mathbf{e}_3 \wedge \mathbf{e}_2)(-\mathbf{e}_3 + 3\mathbf{e}_1 \wedge \mathbf{e}_2) \\
&= (\mathbf{e}_1 + 2\mathbf{e}_3\mathbf{e}_2)(-\mathbf{e}_3 + 3\mathbf{e}_1\mathbf{e}_2) \\
&= 3\mathbf{e}_1\mathbf{e}_1\mathbf{e}_2 + 6\mathbf{e}_3\mathbf{e}_2\mathbf{e}_1\mathbf{e}_2 - \mathbf{e}_1\mathbf{e}_3 - 2\mathbf{e}_3\mathbf{e}_2\mathbf{e}_3 \\
&= 3\mathbf{e}_2 - 6\mathbf{e}_3\mathbf{e}_1\mathbf{e}_2\mathbf{e}_2 + \mathbf{e}_3\mathbf{e}_1 + 2\mathbf{e}_2\mathbf{e}_3\mathbf{e}_3 \\
&= 3\mathbf{e}_2 - 6\mathbf{e}_3\mathbf{e}_1 + \mathbf{e}_3\mathbf{e}_1 + 2\mathbf{e}_2 \\
&= \mathbf{e}_2 - 5\mathbf{e}_3 \wedge \mathbf{e}_1
\end{aligned}$$

De plus, afin de pouvoir retrouver la notion d'orthogonalité, nous définissons un troisième produit, la **contraction à gauche**, notée par '$\rfloor$'. La contraction à gauche est une extension de la forme bilinéaire définie sur $\mathcal{E}$ à tous les poly-vecteurs de l'algèbre géométrique. Elle possède les propriétés suivantes :

$$\begin{array}{ll}
\text{Scalaire et } k\text{-vecteur, } k \neq 0 & \alpha \rfloor \mathbf{B} = \alpha \mathbf{B} \\
 & \mathbf{B} \rfloor \alpha = 0 \\
\text{Vecteurs} & \mathbf{a} \rfloor \mathbf{b} = \mathbf{a} \cdot \mathbf{b} \\
\text{Vecteurs et } k\text{-vecteur} & \mathbf{a} \rfloor (\mathbf{b} \wedge \mathbf{C}) = (\mathbf{a} \rfloor \mathbf{b}) \wedge \mathbf{C} - \mathbf{b} \wedge (\mathbf{a} \rfloor \mathbf{C}) \\
k\text{-vecteurs} & (\mathbf{A} \wedge \mathbf{B}) \rfloor \mathbf{C} = \mathbf{A} \rfloor (\mathbf{B} \rfloor \mathbf{C})
\end{array} \tag{3}$$

La façon d'étendre la forme bilinéaire n'est pas unique. Selon l'extension choisie il sera plus ou moins aisé d'obtenir des algorithmes. Cela est encore le sujet de travaux de recherche [4].

3. Implantation fonctionnelle de l'Algèbre Géométrique en Objective Caml

Dans cette partie, nous présentons deux implantations de l'algèbre géométrique en Objective Caml. La première définit trois modules correspondant à l'algèbre géométrique sur trois espaces vectoriels réels différents. La seconde est un module paramétré définissant de manière générique l'algèbre géométrique en fonction d'un espace (pseudo) euclidien de référence.

Nous ne discuterons ici que de la structure de données et du principe général d'implantation des trois produits principaux, les autres opérations sont obtenues à partir de ceux-ci. De plus amples informations quant à l'ensemble des opérations implantées peuvent être trouvées dans [5].

Toutefois, avant de voir la programmation de l'algèbre géométrique en O'Caml, étudions plus en détail celle en C++ avec Gaigen.

3.1. Présentation du logiciel Gaigen

Gaigen [3] est un générateur de bibliothèques C++ implantant l'algèbre géométrique pour des espaces euclidiens ou pseudo-euclidiens de dimension inférieures à 8.

Il inclut une optimisation sur la structure de données, en représentant les poly-vecteurs uniquement par l'information utile, c'est-à-dire en ne représentant uniquement les coordonnées correspondants aux parties k-vectorielles non nulles. Par exemple, un 1-vecteur de $\mathcal{G}_3$ est un poly-vecteur ayant ses trois parties de grades 0, 2 et 3 nulles, on ne le représente donc que par ses trois coordonnées dans la base de $\mathcal{G}_3^1$. Ceci permet de réduire l'utilisation de la mémoire, ainsi que les temps de calculs.

La structure de données est constituée, d'une part d'un tableau dynamique de flottants où seront stockés les coordonnées du poly-vecteur dans la base de l'algèbre, d'autre part d'une variable, appelée `usage`, indiquant à quoi correspondent les coefficients dans le tableau. On peut voir ici que l'avantage de l'optimisation sur la structure est donc limité par la manipulation complexe de la variable `usage`, qui nécessite d'être recalculée à chaque modification du poly-vecteur, de la même façon que le tableau dynamique doit être ré-alloué.

Dés lors, la manipulation de la structure de données définie par Gaigen semble particulièrement inadaptée dans un langage impératif tel que le C++. Toutefois, Gaigen s'avère être pour le moment la réalisation la plus efficace sur l'algèbre géométrique.

3.2. Motivations pour l'utilisation du langage Objective Caml

Notre intérêt pour les langages fonctionnels, en particulier pour le langage Objective Caml, repose sur une remarque simple. Outre la concision et la clarté des programmes écrits en O'Caml, qui sont intéressants dans notre cas car notre code n'est pas généré, ce langage offre un formalisme plus proche des mathématiques. Ainsi, à l'opposé des langages impératifs, le langage Objective Caml est particulièrement approprié pour faire de la manipulation de termes typés complexes, ce que nous faisons précisément avec l'algèbre géométrique.

3.3. Implantation de l'algèbre géométrique sur deux espaces euclidiens et un espace pseudo-euclidien

Nous avons développé trois modules O'Caml correspondant à l'algèbre géométrique sur les trois espaces $\mathbb{R}^3$, $\mathbb{R}^4$ et $\mathbb{R}^{4,1}$. Pour le dernier espace, le couple $(4,1)$ correspond à la signature de l'espace. Le premier nombre correspond à la dimension maximale d'un sous-espace pour lequel les vecteurs sont de signature positive pour la forme quadratique, tandis que le second nombre correspond à la plus grande dimension possible du sous-espace de signature négative. On parle alors d'espace pseudo-euclidien. Chacun de ces espaces est utilisé pour modéliser la géométrie d'un espace affine euclidien à trois dimensions [6, 7].

3.3.1. Structure de données

Du fait de la grande dimension de l'espace de l'algèbre, l'optimisation utilisée par les bibliothèques générées à partir de Gaigen est justifiée. Cependant, nous avons pu constater dans le cas de Gaigen que la structure de données et sa manipulation s'en trouvent notablement complexifiées. L'idée est donc de représenter uniquement l'information utile des poly-vecteurs, mais plus simplement.

La définition d'une structure de données incluant cette optimisation est assez simple en O'Caml. Tout d'abord, nous définissons un type pour chaque grade k, $k \in [0;n]$, où n est la dimension de l'espace d'origine. Pour les scalaires et les n-vecteurs, respectivement pour les grades $k=0$ et $k=n$, le type est un alias du type `float`, les espaces $\mathcal{G}_n^0$ et $\mathcal{G}_n^n$ étant chacun de dimension 1. Pour les poly-vecteurs de grade k, $k \in [1;n-1]$, les types sont des enregistrements contenant exactement $\binom{n}{k} = \dim \mathcal{G}_n^k$ champs de type `float`. L'exemple de la structure pour $\mathcal{G}_3$ est donné ci-dessous.

```
type scalar = float
type vector = {e1 : float ; e2 : float ; e3 : float ; }
type bivector = {e23 : float ; e31 : float ; e12 : float ; }
type trivector = float
```

Chaque donnée de type `float` correspond alors à une coordonnée du poly-vecteur dans la base de l'algèbre. Évidemment, ces types changent selon la dimension de l'espace euclidien ou pseudo-euclidien de départ.

Une fois ces structures déclarées, nous définissons un second type nommé `n_blade`, qui représente un k-vecteur pour tout k, $0 \leq k \leq n$. Il s'agit d'un type somme permettant de représenter indifféremment, un des types définis ci-dessus.

```
type n_blade =
    Scalar of scalar
  | Vector of vector
  | Bivector of bivector
  | Trivector of trivector
type multi_vector = n_blade list
```

Enfin, un poly-vecteur est représenté par une liste de `n_blade`, représentant implicitement l'opération d'addition entre les parties k-vectorielles du poly-vecteur, pour $k \in [0;n]$.

Pour faciliter les traitements sur la structure, nous avons défini des contraintes de cohérence sur celle-ci telles que le tri selon le grade des listes de `n_blade` et l'unicité dans la liste de la partie d'un grade donné. Ces contraintes sont maintenues par la construction, dans les opérations de l'algèbre.

On remarque ici qu'il n'est nul besoin de structure supplémentaire pour manipuler les poly-vecteurs, comme cela est fait dans les bibliothèques produites par Gaigen avec la variable `usage`. L'avantage est que la création et la modification d'un poly-vecteur ne nécessitent pas d'opérations supplémentaires, la manipulation de la structure est immédiate. Cet avantage est mis à profit dans la programmation des opérations comme nous allons le voir maintenant.

3.3.2. Principe d'implantation des produits de l'algèbre

L'implantation des lois internes de l'algèbre, c'est-à-dire l'addition et le produit géométrique, est réalisée à partir de leurs propriétés algébriques. Il en va de même pour les deux produits dérivés du produit géométrique. Concernant les produits géométrique, interne et externe, la propriété qui est mise à profit est la bilinéarité. Cela permet de faire des traitements sur chaque partie k-vectorielle séparément des autres. Pour réaliser cela, nous procédons par filtrages successifs sur les éléments de la liste, dans un premier temps sur le premier argument, puis dans un second temps sur le deuxième argument (voir listing 1).

```
let rec ( *: ) multi_vector1 multi_vector2 =
  let rec gp_sc scalar multi_vector2 = ...
  and gp_vec vector multi_vector2 = ...
  and gp_bivec bivector multi_vector2 = match multi_vector2 with
        [] -> []
      | (Scalar scalar2)::tl -> ([Bivector { ... }]) +: (gp_bivec bivector tl)
      | (Vector vector2)::tl ->
          ((Vector {e1 = bivector.e12 *. vector2.e2 -. bivector.e31 *. vector2.e3;
                    e2 = bivector.e23 *. vector2.e3 -. bivector.e12 *. vector2.e1;
                    e3 = bivector.e31 *. vector2.e1 -. bivector.e23 *. vector2.e2;})::
          (Trivector (bivector.e23 *. vector2.e1 +. bivector.e31 *. vector2.e2 +.
                      bivector.e12 *. vector2.e3))::[]) +: (gp_bivec bivector tl)
      | (Bivector bivector2)::tl -> ( ... ) +: (gp_bivec bivector tl)
      | [Trivector trivector2] -> [Vector { ... }]
      | _ -> raise Pseudoscalar_not_at_end_of_Multivector
  and gp_trivec trivector b = ...
  in match multi_vector1 with
      [] -> []
    | (Scalar scalar) :: tail ->
        (gp_sc scalar multi_vector2) +: (tail *: multi_vector2)
    | (Vector vector) :: tail ->
        (gp_vec vector multi_vector2) +: (tail *: multi_vector2)
    | (Bivector bivector) :: tail ->
        (gp_bivec bivector multi_vector2) +: (tail *: multi_vector2)
    | (Trivector trivector) :: tail ->
        (gp_trivec trivector multi_vector2) +: (tail *: multi_vector2)
```

Listing 1: Extrait du produit géométrique sur $\mathcal{G}_3$

Le produit est ainsi décomposé par un produit entre deux poly-vecteurs de grade unique. Il suffit alors de déterminer le résultat du produit entre un i-vecteur et un j-vecteur, $(i,j) \in [0;n] \times [0;n]$, pour savoir quelles opérations effectuer sur les flottants contenus dans la structure.

Le même principe de décomposition de la liste est utilisé pour l'opération d'addition de poly-vecteurs.

Ce que l'on peut noter ici est que la programmation des opérations et la manipulation de la structure est simplifiée par l'utilisation du filtrage de motifs sur notre structure de données. En comparaison aux bibliothèques générées par Gaigen, le code produit est plus simple, plus court et plus lisible, celui-ci étant plus proche de la définition des règles et propriétés algébriques des entités manipulées.

3.4. Implantation de l'algèbre paramétrée par un espace

Comme rappelé dans la partie 2, l'algèbre géométrique est définie à partir d'un espace euclidien ou pseudo-euclidien. Dans la partie 3.3, nous avons effectivement vu que l'implantation des trois modules utilise une structure de données semblable. L'idée est alors d'utiliser les modules paramétrés du langage Objective Caml pour programmer un module générique de l'algèbre géométrique en fonction d'un espace (pseudo) euclidien.

L'intérêt d'un tel module est, comme dans le cas de Gaigen, d'obtenir rapidement et simplement une implantation de l'algèbre géométrique à partir de la spécification d'un espace. Cependant, dans le cas de Gaigen, les opérations ne sont pas génériques, mais sont implantées de manière *ad hoc* à partir de matrices donnant le résultat d'un produit en fonction de la base de l'espace. Les bibliothèques générées sont alors limitées à des algèbres pour des espaces de dimension inférieures à huit. Ce n'est pas le cas de notre module paramétré, où les fonctions sont génériques et où les résultats sont déduits de manière automatique, quelle que soit la dimension de l'espace donné en paramètre.

3.4.1. Spécification de l'espace

Un espace euclidien ou pseudo-euclidien est défini par une base canonique, indiquant la dimension de l'espace, et par une forme quadratique, donnant la signature de l'espace. La base étant orthonormée, la forme bilinéaire est nulle pour deux vecteurs différents dans la base. Le listing 2 montre d'une part la signature du module O'Caml d'un espace, et d'autre part la structure d'un module pour l'espace pseudo-euclidien $\mathbb{R}^{4,1}$.

```
module type SPACE =
  sig
    type canonic_vector
    type sign = LESS | NULL | PLUS
    val canonic_basis : canonic_vector list
    val _Q : canonic_vector -> sign
    val string_vector : canonic_vector -> string
  end;;

module C3D_SPACE : SPACE =
  struct
    type canonic_vector = E1 | E2 | E3 | EP | EM
    type sign = LESS | NULL | PLUS
    let canonic_basis = [E1; E2; E3; EP; EM]
    let _Q = function EM -> LESS | _ -> PLUS
    let string_vector vector = match vector with
      E1 -> "e1" | E2 -> "e2" | E3 -> "e3" | EP -> "e+" | EM -> "e-"
  end;;
```

Listing 2: Signature et structure de la spécification de l'espace vectoriel $\mathbb{R}^{4,1}$

La base canonique de l'espace est définie par le type somme `canonic_vector` comprenant cinq constructeurs constants, correspondant au cinq vecteurs orthonormés de la base. Il n'est pas nécessaire d'associer une valeur à ces constructeurs et donc à ces vecteurs, il suffit de leur donner une *existence* et de les différencier.

Le type `sign` fixe les trois valeurs possibles prisent par la forme quadratique sur les vecteurs de la base canonique et permet d'indiquer la signature de l'espace. `LESS` correspond à une signature négative (-1), `NULL` à une signature nulle (0) et `PLUS` à une signature positive $(+1)$. Ce type est complètement spécifié dans la signature de l'espace, afin que l'on ne puisse pas le redéfinir, celui-ci étant utilisé dans la structure du module paramétré de l'algèbre géométrique. La liste `canonic_basis` permet au module paramétré générant l'algèbre géométrique, de connaître et de manipuler les vecteurs de la base de l'espace ; elle doit contenir tous les vecteurs de cette base. La fonction `_Q`, quant à elle, représente la définition de la forme quadratique définie sur l'espace, restreinte aux vecteurs de la base canonique. Cela suffit à déterminer la valeur de la forme bilinéaire pour tous vecteurs de l'espace, ceux-ci s'écrivant comme combinaisons linéaires des vecteurs de la base. La fonction `string_vector` est une fonction additionnelle qui permet de faire un affichage des poly-vecteurs, en associant une chaîne de caractères à chaque vecteur de la base.

3.4.2. Structure de données utilisée

Du fait de la généricité de ce module, il est nécessaire de définir une structure de données générale, indépendante de la dimension de l'espace. De ce fait, il est impossible de reprendre une structure similaire à celle des modules non paramétrés, décrite dans la partie 3.3.

Nous redéfinissons le type `n_blade` par un couple contenant un flottant et une liste de vecteurs de la base de l'espace (voir ci-dessous). La relation implicite associant les différents éléments de cette liste est le produit externe. Par exemple, en considérant la base de l'espace spécifié dans le listing 2, la liste `[E2; E3; EM]` correspond au trivecteur $\mathbf{e}_2 \wedge \mathbf{e}_3 \wedge \mathbf{e}_-$. Cette représentation reste cohérente du fait que le produit externe est associatif. Le flottant associé à la liste dans le type `n_blade` correspond au coefficient associé au poly-vecteur de la base de l'algèbre représenté par la liste. Le couple correspond alors à un monôme de l'algèbre.

```
module GEOMETRIC_ALGEBRA : GEOMETRIC_ALGEBRA_SIG =
  functor (Model : SPACE) ->
  struct
    type n_blade = float * Model.canonic_vector list
    type multi_vector = n_blade list
...
```

La représentation générale d'un poly-vecteur vient alors naturellement comme une liste de `n_blade`. Ici encore, le constructeur de liste correspond à l'addition de l'algèbre.

Pour définir les contraintes de cohérence de la structure, nous utilisons une relation d'ordre total sur les listes contenues dans le type `n_blade`. Cette relation est définie d'abord sur la longueur de celles-ci, de sorte à trier les monômes sur le grade, puis sur l'ordre des éléments de la liste de type `canonic_vector`, dont un ordre total sur ce type est fixé par le langage.

Toutefois, du fait de l'antisymétrie et de l'associativité du produit externe sur les vecteurs de l'espace (pseudo) euclidien, deux listes contenant les mêmes vecteurs de la base canonique mais dans un ordre différent représentent le même poly-vecteur, à une permutation près et, par conséquent, au signe près. C'est le cas, par exemple, des listes `[E1; EP; E3]` et `[E3; E1; EP]`. Pour déterminer une base de l'algèbre, nous imposons un tri sur les listes par ordre croissant des vecteurs. Pour l'exemple précédent, les deux listes doivent donner la liste `[E1; E3; EP]` après le tri. Le signe du coefficient associé à la liste dans le couple est alors changé selon le nombre de permutations nécessaires pour trier la liste. De même, si lors du tri on détecte une liste de vecteurs contenant deux fois un même élément, la relation $\forall \mathbf{a} \in \mathcal{E}, \mathbf{a} \wedge \mathbf{a} = 0$ indique que le poly-vecteur est nul et qu'il doit donc être retiré de la liste `multi_vector`.

Enfin, grâce à la relation d'ordre total définie sur les éléments de type `n_blade`, nous imposons encore un tri croissant de la liste de type `multi_vector`. La liste obtenue est triée de façon strictement croissante. En effet, comme une liste de type `multi_vector` représente une addition de monômes, si deux couples de type `n_blade` contiennent les mêmes listes, ceux-ci peuvent être aisément fusionnés en additionnant les deux coefficients.

Pour résumer, en reprenant l'exemple précédent avec la liste suivante :

```
[(2.,[E1;EP;E3]); (1.3,[E4;E2;E4]); (-.4.5,[E3;E1;EP])].
```

La cohérence de la structure de données définie ci-dessus permet de réduire cette liste à la liste ci-après :

```
[(2.5,[E1;E3;EP])]
```

3.4.3. Implantation des produits de l'algèbre

Remarquons que les définitions du produit géométrique et du produit interne de l'algèbre sont directement dépendantes de la forme bilinéaire de l'espace euclidien ou pseudo-euclidien. Cette forme bilinéaire est définie dans le module par la fonction `|.` ci-dessous. Cette forme est décrite en fonction de sa forme quadratique associée `_Q`, définie par l'espace donné en paramètre du module.

```
let ( |. ) vector1 vector2 =
  if (vector1 = vector2)
  then (match (Model._Q vector1) with
    Model.LESS -> -.1. | Model.NULL -> 0. | Model.PLUS -> 1.)
  else 0.
```

L'implantation des produits est directement inspirée par les propriétés de ceux-ci. Comme dans le cas des modules non paramétrés, nous utilisons la bilinéarité pour décomposer les deux arguments par filtrages successifs sur les éléments de la liste. Ensuite, les traitements faits sur les monômes sont eux aussi déduits des propriétés algébriques des produits respectifs, présentées dans la partie 2.

```
let ( *: )  multi_1 multi_2
  let rec permut blade =
    let rec order blade permut = match blade with
      (f, []) -> ((f, []), permut)
    | (f, vector::[]) -> ((f, vector::[]), permut)
    | (f, vector1::vector2::tail) ->
        if (vector1 = vector2)
        then (order (f *. (vector1 |. vector2), tail) permut)
        else (if (vector1 < vector2)
          then (let ((ff, blade), n) = (order (f, vector2::tail) permut)
            in ((ff, vector1::blade), n))
          else (let ((ff, blade), n) = (order (-. f, vector1::tail) (permut + 1))
            in ((ff, vector2::blade), n)))
    in let (blade_aux, nb_perm) = (order blade 0)
    in if (nb_perm = 0) then blade_aux else (permut blade_aux)
    and distrib2 (f1, blade1) multi = match multi with
        [] -> []
      | (f2, blade2)::tail -> (permut (f1*.f2, blade1@blade2))::
                              (distrib2 (f1, blade1) tail)
    and distrib1 multi_1 multi_2 = match multi_1 with
        [] -> []
      | blade::tail -> (distrib2 blade multi_2) +: (distrib1 tail multi_2)
    in reduce (distrib1 multi_1 multi_2)
```

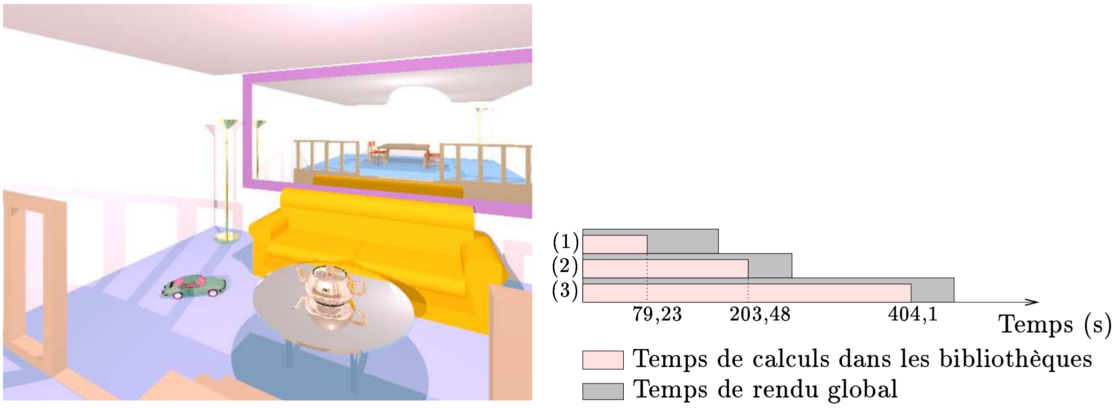
Listing 3: Produit géométrique de l'algèbre paramétrée

Le listing 3 présente l'exemple de l'implantation du produit géométrique. La décomposition du poly-vecteur en monômes de l'algèbre se fait dans `distrib1` pour le premier argument et dans `distrib2` pour le second. Les propriétés des produits sont alors utilisées dans les fonctions `permut` et `order`, qui trient la liste puis la réduisent selon les propriétés du produit.

Ainsi, comme on peut le voir, le développement d'un module générique est facilité par l'utilisation d'un langage fonctionnel tel que O'Caml, ceci grâce à la correspondance directe qu'il est possible de créer entre les opérations du langage (constructeurs de types) et les opérations de l'algèbre (addition, produit externe, etc.). La programmation se fait alors à partir des définitions des opérations de l'algèbre. Cependant, la généricité nécessite la définition d'une structure plus complexe qui, bien qu'assez simple à utiliser, nécessite un grand nombre d'opérations pour maintenir sa cohérence, ce qui va se ressentir sur la vitesse d'exécution.

4. Comparaison de l'efficacité des bibliothèques

Pour comparer l'efficacité de nos bibliothèques avec les bibliothèques générées par Gaigen, nous avons utilisé un lancer de rayons. Le lancer de rayons est une des premières méthodes utilisées en synthèse d'images, introduite par Whitted [8], pour obtenir une visualisation réaliste d'une scène constituée de primitives géométriques (polygones, sphères, cylindres, etc.), de lumières et d'une caméra.



(a) Une scène de test (19594 triangles, 271 polygones)

(b) Temps de rendu de la scène en secondes : (1) bibliothèque O'Caml ; (2) Gaigen, bibliothèque optimisée ; (3) Gaigen, bibliothèque non optimisée.

Figure 2: Efficacité des bibliothèques dans un lancer de rayons

L'intérêt d'un tel logiciel pour tester notre bibliothèque, tient dans sa simplicité de mise en œuvre ainsi que son besoin en rapidité et en efficacité. Dans notre cas, nous nous sommes intéressés au calcul d'intersections entre un rayon et un nombre restreint de primitives géométriques, pour lesquels l'algèbre géométrique offre une formulation particulièrement simple. Ces primitives sont d'une part le triangle et le polygone quelconque, pour lesquels nous utilisons l'algèbre $\mathcal{G}_4$, qui correspond à l'algèbre du modèle projectif ou homogène [6], et d'autre part la sphère, pour laquelle nous utilisons l'algèbre $\mathcal{G}_{4,1}$, qui correspond à l'algèbre du modèle conforme [9].

Nous avons effectué la comparaison de deux des modules O'Caml avec, pour chacun d'eux, deux bibliothèques générées par Gaigen, l'une incluant des optimisations sur les produits, l'autre, n'utilisant aucune autre optimisation que celle de la structure de données. Le lancer de rayon est écrit en C ; cela signifie que nous avons du interfacer les langages O'Caml/C puis C++/C.

Comme le montre la figure 2(b) sur une scène de test pour l'intersection avec des triangles (figure 2(a)), notre bibliothèque en Objective Caml est plus rapide que la bibliothèque C++ optimisée. Le temps nécessaire au calcul d'intersections avec l'algèbre géométrique varie d'un facteur d'environ 2,5 en faveur de notre bibliothèque. On remarque cependant que le temps nécessaire à l'interfaçage des langages est beaucoup plus important en O'Caml (différences entre les zônes sombres). L'étude du profil de l'exécution, dont un extrait est présenté dans la table 1, montre que les modules C++ perdent principalement du temps dans une fonction de copie de tableaux de flottants (fonction `p3gai__copy`, 110,11s pour la bibliothèque non optimisée et 66,44s pour celle optimisée), contenant les coordonnées des poly-vecteurs, puis dans la fonction `p3gai::setUsage(int)` qui recalcule la valeur de la variable `usage` qui gère le contenu du tableau dynamique (respectivement 51,37s et 49,11s). Dans le cas de la bibliothèque O'Caml, la fonction la plus coûteuse est une fonction interne au lancer de rayons (`SpatialIntersect`) qui arrive seulement en cinquième et troisième position respectivement pour les bibliothèques C++ non optimisée et optimisée, pour des temps similaires (environ 25s), tandis que deux fonctions d'interfaçage des langages (`vector2value` et `copy_double`) sont parmi les cinq fonctions les plus coûteuses. Ceci montre clairement l'avantage du langage O'Caml sur le C++ pour programmer l'algèbre géométrique.

Bibliothèque O'Caml sur $\mathcal{G}_4$				Bibliothèque C++ optimisée sur $\mathcal{G}_4$			
% time	cumulative seconds	self seconds	name	% time	cumulative seconds	self seconds	name
15.42	25.74	25.74	SpatialIntersect	25.80	66.44	66.44	p3gai__copy
11.50	44.94	19.20	GEOMETRIC_ALGEBRA__op_vec_372	19.07	115.55	49.11	p3gai::setUsage(int)
6.20	55.28	10.35	vector2value	9.60	140.27	24.72	SpatialIntersect
5.05	63.72	8.43	GEOMETRIC_ALGEBRA__$2b$3a_177	6.25	156.37	16.10	p3gai::copy(p3gai const&)
4.98	72.03	8.31	copy_double	4.49	167.94	11.57	p3ga::~p3ga()
4.26	79.14	7.11	GEOMETRIC_ALGEBRA__$5e$3a_368	2.57	174.56	6.62	p3gai_opt_02_op_02

Table 1: Extraits de deux profils d'exécution

Nous avons aussi testé le module paramétré sur une scène simple (271 polygones). Comme nous le présagions, il reste inefficace pour ce qui est de la vitesse d'exécution ; en effet, il est environ quinze fois moins rapide que la bibliothèque C++ non optimisée (654,32s contre 42,98s) pour la scène de test. Toutefois, ce module reste intéressant pour obtenir rapidement une première implantation de l'algèbre géométrique d'un espace de grande dimension.

5. Conclusion

Dans cet article, nous avons montré qu'utiliser un langage fonctionnel rend plus facile la programmation de l'algèbre géométrique. Le langage Objective Caml s'avère effectivement plus apte à définir la structure de données et les opérations manipulant cette structure que le langage C++. Cette aptitude se ressent aussi en pratique, par une plus grande rapidité à l'exécution.

Désormais, notre motivation est d'appliquer l'algèbre géométrique aux problèmes de géométrie qui interviennent en synthèse d'images et notamment dans les simulateurs de propagation d'ondes. Cependant, le coût de l'interfaçage entre le langage O'Caml et le langage C est assez conséquent et nous fait perdre un peu de l'efficacité du langage. Il est donc utile de connaître dans quelle proportion il est intéressant d'utiliser le langage O'Caml dans le développement d'applications en synthèse d'images, et ainsi d'éviter l'interfaçage coûteux.

Une autre utilisation possible des langages fonctionnels avec l'algèbre géométrique serait d'utiliser un principe *d'évaluation paresseuse* pour réduire les termes induits par une opération spécifique sur les poly-vecteurs. Cela réduirait ainsi les calculs sur les coordonnées et donc à la fois les erreurs sur les flottants et les temps d'exécution.

Concernant le module paramétré, bien que peu efficace en temps d'exécution, nous envisageons de l'utiliser afin de générer le code de modules optimisés, comme le fait Gaigen. Dans ce cas, nous pourrions envisager une structure de données plus spécifique et efficace que nous n'avons pu considérer en raison du code long et répétitif qu'elle impliquerait.

Bibliographie

[1] David Hestenes. *Spacetime Algebra.* Gordon & Breach, 1966.

[2] C. Perwasss D. Hildenbrand, D. Fontijne and L. Dorst. Geometric algebra and its application to computer graphics. Tutorial 3, 25^{th} Annual Conference of the European Association of Computer Graphics, September 2004.

[3] Tim Bouma Daniel Fontijne, Leo Dorst. Gaigen : A geometric algebra implementation generator. http://www.science.uva.nl/ga/gaigen/index.html, 2003. University of Amsterdam.

[4] Leo Dorst. The inner products of geometric algebra. In Lasenby, Dorst, Doran, eds, *Applications of Geometric Algebra in Computer Science and Engineering*, chapter 2. Birkhauser, 2002.

[5] S. Charneau. Algèbre Géométrique et Lancer de Rayons. Mémoire de DEA, Université de Poitiers, Juin 2004.

[6] Hongbo Li & Alyn Rockwood David Hestenes. Generalized homogeneous coordinates for computational geometry. In G. Sommer, editor, *Geometric Computing with Clifford Algebra*, pages 25–58. Springer-Verlag Heidelberg, 2000.

[7] David Hestenes. Old wine in new bottles : A new algebraic framework for computational geometry. In E. Bayro-Corrochano & G. Sobczyk, editor, *Advances in Geometric Algebra with Applications in Science and Engineering*, pages 1–14. Birkhauser, 2001.

[8] T. Whitted. An improved illumination model for shaded display. In *Communications of the ACM*, pages 343–349. Communications of the ACM, Juin 1980.

[9] Anthony Lasenby Chris Doran and Joan Lasenby. Conformal geometry, euclidean space and geometric algebra. In J. Winkler ed, editor, *Uncertainty in Geometric Computations.* Kluwer, 2002.