La géométrie dans l'espace avec METAPOST

Denis Roegel

LORIA
BP 239
54506 Vandœuvre-lès-Nancy cedex
roegel@loria.fr

Résumé. METAPOST est un outil particulièrement bien adapté à la réalisation des dessins techniques d'un document. Dans cet article, nous montrons comment METAPOST peut être utilisé pour représenter des objets dans l'espace, et plus particulièrement le profit que l'on peut en tirer pour faire des constructions géométriques faisant intervenir des droites, des plans, ainsi que leurs intersections, plans orthogonaux, etc. L'ensemble des fonctionnalités est regroupé sous la forme d'un nouveau module METAPOST qui s'adresse autant à ceux qui enseignent qu'à ceux qui étudient la géométrie.

Abstract. *METAPOST is a tool especially well-suited for the inclusion of technical drawings in a document. In this article, we show how METAPOST can be used to represent objects in space and especially how it can be used for drawing geometric constructions involving lines, planes, as well as their intersections, orthogonal planes, etc. All the features belong to a new METAPOST package aimed at all those who teach and study geometry.*

Cet article est dédié à Donald Knuth dont la thèse portait sur la géométrie projective.

1. Introduction

METAPOST [5, 2, 6, 4] est un langage de description graphique créé par John Hobby à partir du système METAFONT de Donald Knuth [7]. Avec METAPOST, un dessin dans un espace à deux dimensions est représenté sous forme de programme qui est compilé en fichier PostScript. Un dessin peut être décrit de manière très précise et compacte en tirant profit de la nature déclarative du langage. Ainsi, les contraintes linéaires entre les coordonnées de plusieurs points, comme par exemple une symétrie centrale, sont exprimées très naturellement sous forme d'équations. De plus, METAPOST permet de manipuler

des équations faisant intervenir des quantités qui ne sont pas complètement déterminées. Par exemple, pour dire que le point p_3 se trouve au milieu du segment $[p_1, p_2]$, il suffit d'écrire : `p3-p1=p2-p3` ou bien `p3=.5[p1,p2]`.

Lorsque cette équation apparaît, il se peut qu'aucune des coordonnées des trois points ne soit connue. La prise en compte de l'équation conduit à l'ajout d'une contrainte. Ces contraintes se rajoutent jusqu'à déterminer précisément les quantités mises en jeu. Dans l'exemple précédent, les trois points peuvent être entièrement déterminés en positionnant précisément p_1 et p_2. Une valeur peut rester indéterminée tant qu'elle n'intervient pas dans un tracé. Enfin, METAPOST signale si certaines équations sont redondantes ou inconsistantes.

2. Un premier exemple en géométrie plane

Pour bien comprendre comment METAPOST permet naturellement d'exprimer un problème géométrique, voyons par exemple la représentation d'une propriété du triangle, telle que l'existence du *cercle des neuf points* (affirmée pour la première fois par Poncelet et Brianchon en 1821). La figure 1 montre le résultat produit par METAPOST. Ce premier exemple servira en même temps d'introduction à METAPOST pour le lecteur qui découvrirait ici ce langage.

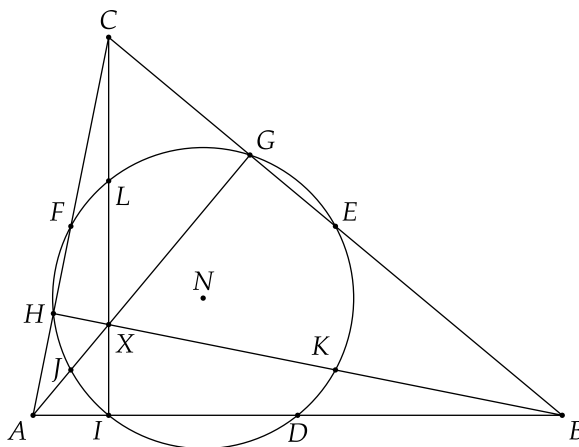


FIGURE 1 – Le cercle des neuf points.

Dans cette figure, nous avons commencé par définir les points, puis placer les trois sommets du triangle en fonction de l'origine (`origin`) et d'une unité

arbitraire u qui nous permet de changer la taille du dessin facilement par la suite [1] :

```
numeric u; u=1cm;
pair A,B,C,D,E,F,G,H,I,J,K,L,N,X;
A=origin; B-A=(7u,0); C-A=(u,5u);
```

Ensuite, les milieux D, E et F des côtés du triangle sont déterminés par des équations :

```
D=.5[A,B]; E=.5[B,C]; F=.5[A,C];
```

G, H et I sont les bases des hauteurs du triangle. Elles peuvent être obtenues simplement en calculant l'intersection d'un côté avec un segment partant du sommet opposé et dirigé suivant une direction à angle droit de celle du côté considéré. La définition `whatever` est particulièrement utile dans ce cas, puisqu'elle représente une inconnue sans nom (de sorte que plusieurs occurrences de `whatever` ne désignent pas la même inconnue !). Nous avons donc :

```
G=whatever[B,C]=whatever[A,A+((C-B) rotated 90)];
```

Cela signifie que G est quelque part sur la droite (BC) et aussi quelque part sur la droite (AP) où P est un point de la hauteur. METAPOST va donner une valeur aux *deux* inconnues pour satisfaire cette équation. De manière similaire,

```
H=whatever[A,C]=whatever[B,B+((C-A) rotated 90)];
I=whatever[A,B]=whatever[C,C+((B-A) rotated 90)];
```

L'orthocentre (intersection des hauteurs) s'obtient avec `intersectionpoint`. La construction `A--G` représente le segment $[AG]$:

```
X=(A--G) intersectionpoint (C--I);
```

Les milieux J, K et L des segments $[AX]$, $[BX]$ et $[CX]$ s'obtiennent comme précédemment D, E et F :

```
J=.5[A,X]; K=.5[B,X]; L=.5[C,X];
```

1. Pour les points, nous aurions aussi pu utiliser `z0`, `z1`, etc., qui sont des variables prédéfinies.

Enfin, pour obtenir le centre N du cercle des neufs points (à supposer que ce cercle existe), il suffit de calculer l'intersection de deux médiatrices, par exemple celles des segments $[ID]$ et de $[DH]$:

```
N=whatever[.5[D,I],(.5[D,I]+((D-I) rotated 90))]
 =whatever[.5[D,H],(.5[D,H]+((D-H) rotated 90))];
```

Le rayon du cercle est déterminé de la manière suivante :

```
r=arclength(I--N);
```

Le triangle ainsi que les hauteurs et le cercle (de centre N et de diamètre $2r$) sont tracés avec :

```
draw A--B--C--cycle; draw A--G; draw B--H; draw C--I;
draw fullcircle scaled 2r shifted N;
```

Les points sont marqués avec `drawdot` après avoir épaissi le trait. Enfin, les annotations sont toutes faites sur le modèle de :

```
label.top(btex $C$ etex,C);
```

L'instruction `label` permet d'inclure des étiquettes TEX.

Cet exemple est révélateur de l'expression naturelle des contraintes géométriques pour les problèmes de géométrie plane. Toutes les constructions que nous venons d'utiliser sont absolument standard dans METAPOST. Bien sûr, si nous avions beaucoup de telles figures, nous introduirions des fonctions pour le calcul des hauteurs, des médiatrices, etc.

3. Extensions de METAPOST

METAPOST est un système extensible. Il s'agit à la base d'un programme exécutable qui charge un ensemble initial de macros. Il est ensuite possible d'ajouter de nouvelles définitions, propres à un domaine. Ainsi, lorsque nous nous sommes intéressé par le passé à la représentation dans le plan d'objets de l'espace, nous avons développé un *package* `3d`, initialement dans le but de manipuler des polyèdres [11]. Nous avons récemment développé d'autres extensions qui s'appuient sur le *package* `3d`. Nous considérons ces extensions comme

des « modules » du *package* `3d`. En particulier, nous avons voulu manipuler d'autres objets que les polyèdres, comme par exemple des courbes définies par des équations ou encore données par des suites de points. Parmi les extensions créées, nous avons considéré un module fournissant diverses fonctionnalités adaptées à la géométrie dans l'espace. Ce module, que nous introduisons ici, est le module `3dgeom` [2] (voir figure 2).

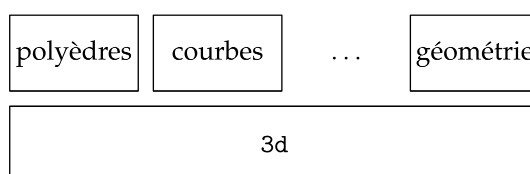


FIGURE 2 – Structure du *package* `3d` et de ses modules.

Certains modules en chargent automatiquement d'autres. Le module `3dgeom` charge ainsi `3d`. Un module ne sera chargé qu'une fois.

Un programme utilisant `3dgeom` commencera donc par `input 3dgeom`.

4. La géométrie dans l'espace

4.1. Un exemple simple

Nous allons entrer en matière en représentant un objet de l'espace très simple à définir, le cube (figure 3). Pour ce faire, nous allons donner les coordonnées de ses huit sommets.

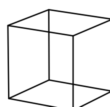


FIGURE 3 – Un cube vu en perspective linéaire.

Le *package* `3d` définit une notion de point ou de vecteur sous forme de triplets de valeurs numériques. Les points doivent être définis par un mécanisme d'allocation et doivent être libérés lorsqu'ils ne servent plus. L'allocation d'un

2. Ce module est disponible sur CTAN en `graphics/metapost/macros/3d`.

point (resp. d'un vecteur) se fait avec `new_point` (resp. `new_vec`). Il s'agit d'une macro prenant en paramètre un nom de point (resp. vecteur) et qui alloue une zone mémoire pour le stocker. La libération d'un point ou d'un vecteur se fait avec `free_point` ou `free_vec`, en donnant l'identificateur du point ou du vecteur en paramètre. Donc, une zone d'un programme souhaitant utiliser un vecteur `v` se présentera ainsi :

```
new_vec(v);
...
free_vec(v);
```

L'ensemble des points et des vecteurs est stocké de manière interne sous la forme d'une pile de triplets. L'allocation ou la libération modifient simplement le pointeur de pile. Il en découle que les points ou les vecteurs doivent être libérés dans l'ordre inverse de leur allocation. Si cet ordre n'est pas respecté, une erreur de désallocation est produite.

```
new_point(pa); new_point(pb);
...
free_point(pb); free_point(pa);
```

On peut, si on le souhaite, ne pas libérer un vecteur ou un point, mais ce genre de manipulation aura souvent des conséquences fâcheuses si les allocations se trouvent au sein de boucles.

Afin de faciliter la manipulation d'ensembles de points, des tableaux de points peuvent être alloués avec `new_points` et libérés avec `free_points`. Un tableau défini ainsi a un nom et un nombre d'éléments n. Les éléments sont indicés de 1 à n. Dans notre exemple, pour créer un cube, nous allons déclarer un tableau `sommet` de huit points :

```
new_points(sommet)(8);
...
free_points(sommet)(8);
```

Chaque sommet peut alors être déclaré avec la commande `set_point_`, par exemple :

```
set_point_(sommet1)(0,0,0);
```

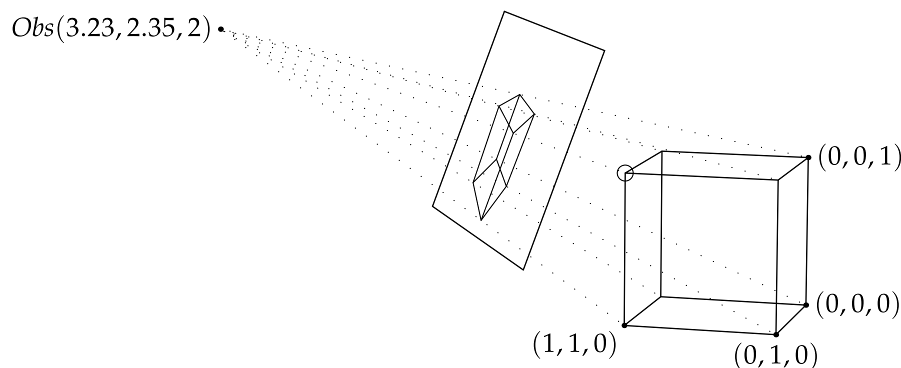


FIGURE 4 – Un cube et sa projection sur l'écran. Le point visé est cerclé.

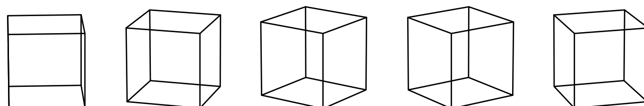


FIGURE 5 – Cinq vues de l'« animation » du cube.

Par défaut, nous sommes en perspective linéaire (ou centrale) et nous avons une notion d'œil ou d'observateur témoin de la scène (figure 4). Celui-ci doit aussi être situé. L'observateur correspond au point prédéfini *Obs*. Sa position peut être définie dans l'espace avec `set_point_`. Un déplacement de l'observateur est obtenu en exprimant ses coordonnées sous forme paramétrée, par exemple [3] :

```
set_point_(Obs)(20*cosd(3.6*i),20*sind(3.6*i),6);
```

En faisant varier i, nous faisons parcourir à l'observateur des points du cercle d'équation $\mathcal{C}(t) = (20\cos(3.6t), 20\sin(3.6t), 6)$. La figure 5 montre cinq vues de ce parcours, pour $i = 0, 5, 10, 15$ et 20. Une animation peut être obtenue en créant un nombre suffisant de vues assez proches et en transformant les sorties METAPOST en fichiers GIF. Le détail de la procédure est expliqué dans notre premier article [11].

Outre la position de l'observateur, il faut définir l'orientation de son regard. Il peut être spécifié à l'aide de trois angles, mais aussi plus simplement en

3. Dans le code, `cosd` et `sind` représentent les fonctions trigonométriques avec des arguments en degrés.

```
input 3danim; drawing_scale:=10cm;
new_points(sommet)(8);
for i:=0 upto 20:
  beginfig(100+i);
    % Cube
    set_point_(sommet1)(0,0,0); set_point_(sommet2)(0,0,1);
    set_point_(sommet3)(0,1,0); set_point_(sommet4)(0,1,1);
    set_point_(sommet5)(1,0,0); set_point_(sommet6)(1,0,1);
    set_point_(sommet7)(1,1,0); set_point_(sommet8)(1,1,1);
    % Observateur
    set_point_(Obs)(20*cosd(3.6*i),20*sind(3.6*i),6);
    Obs_phi:=90; Obs_dist:=2;
    point_of_view_abs(sommet8,Obs_phi);
    % Projections
    for j:=1 upto 8: project_point(j,sommet[j]);
    endfor;
    % Lignes
    draw z1--z2--z4--z3--cycle; draw z5--z6--z8--z7--cycle;
    draw z1--z5; draw z2--z6; draw z3--z7; draw z4--z8;
  endfig;
endfor;
free_points(sommet)(8);
end.
```

FIGURE 6 – Programme produisant l'animation du cube.

indiquant un point de visée et un angle. Pour que l'observateur regarde en permanence le sommet 8, avec l'angle 90 (cet angle est le degré de liberté correspondant à une rotation autour de la droite de visée), il suffit d'écrire :

```
Obs_phi:=90; point_of_view_abs(sommet8,Obs_phi);
```

L'affectation « := » est employée car elle permet de changer une valeur lorsqu'une variable a déjà une valeur. L'écriture `Obs_phi=90` peut provoquer une erreur si `Obs_phi` a une valeur différente de 90.

Enfin, pour déterminer complètement la vue, il faut définir l'emplacement de l'écran. En perspective linéaire, l'écran est un plan perpendiculaire à la direction de visée. C'est sur lui que l'on projette les points de l'espace. La figure 4 montre que le point visé se trouve au centre de l'écran. L'écran est déterminé par sa distance à l'observateur. Cette distance devrait aussi être celle depuis laquelle la vue calculée sera regardée. Nous prenons par exemple :

```
Obs_dist:=2;
```

Cette valeur et les autres valeurs des coordonnées sont multipliées lors de la projection par `drawing_scale` dont la valeur par défaut est fixée à 2 cm. La valeur ci-dessus de `Obs_dist` correspond donc par défaut à un observateur situé à une distance de 4 cm de l'écran.

Une fois les sommets du cube et l'observateur en place, il ne reste qu'à projeter les points sur l'écran avec la commande `project_point`. À chaque point de l'espace correspond un point du plan. Avec `project_point(3,sommet3)`, le point z_3 du plan est associé au sommet 3 du cube. Lorsque les points sont projetés, il ne reste qu'à les tracer :

```
draw z1--z2--z4--z3--cycle; draw z5--z6--z8--z7--cycle;
draw z1--z5; draw z2--z6; draw z3--z7; draw z4--z8;
```

Le programme complet, avec quelques initialisations supplémentaires ainsi que la boucle permettant de créer une animation, est donné à la figure 6. Cet exemple n'a pas fait appel au module de géométrie.

4.2. Améliorations de l'exemple précédent

Le code source produisant le cube est assez simple, mais on peut facilement imaginer qu'un dessin plus complexe devienne très difficile à gérer, ne serait-ce qu'en raison du grand nombre de points impliqués. De plus, les points n'appartiennent pas forcément aux mêmes objets (on peut avoir un cube, un tétraèdre ou encore d'autres objets présents simultanément) et ils peuvent avoir à subir des traitements différents. C'est dans cet esprit que nous avons introduit dans le *package* `3d` des notions de *classe* et d'*objet* qui permettent de regrouper un certain nombre de points, afin en particulier de pouvoir les manipuler globalement ou d'instancier certaines classes plusieurs fois (cf. [10] pour plus de détails sur la programmation orientée objet). Nous allons donc redéfinir le cube, mais sous forme de classe, puis l'instancier.

Le nouveau code (figure 7) montre la définition d'une classe « C ». Tous les objets de cette classe sont des cubes. La définition de la classe se scinde en trois parties, nécessairement appelées `def_C`, `set_C_points` et `draw_C` :

– la fonction générale de définition est `def_C`; cette fonction prend en paramètre un nom d'objet et l'instancie ; en particulier, cette fonction définit le nombre n de points constituant l'objet (ils devront être numérotés de 1 à n) et appelle la fonction définissant les points ; selon la nature des objets définis, il est possible de procéder à d'autres initialisations ; en particulier, bien que ce ne soit pas le cas ici, l'initialisation peut dépendre du nom de l'objet, c'est-à-dire du paramètre ;

```
input 3danim; drawing_scale:=10cm;

vardef def_C(expr inst)=
  new_obj_points(inst,8); set_C_points(inst);
enddef;

vardef set_C_points(expr inst)=
  set_point(1)(0,0,0); set_point(2)(0,0,1); set_point(3)(0,1,0);
  set_point(4)(0,1,1); set_point(5)(1,0,0); set_point(6)(1,0,1);
  set_point(7)(1,1,0); set_point(8)(1,1,1);
enddef;

vardef draw_C(expr inst)=
  draw_lines(1,2,4,3,1); draw_lines(5,6,8,7,5);
  draw_line(1,5); draw_line(2,6); draw_line(3,7); draw_line(4,8);
enddef;

assign_obj("cube","C");

for i:=0 upto 20:
  beginfig(100+i);
    % Observateur
    set_point_(Obs)(20*cosd(3.6*i),20*sind(3.6*i),6);
    Obs_phi:=90; Obs_dist:=2; point_of_view_obj("cube",8,Obs_phi);
    draw_obj("cube");
  endfig;
endfor;

end.
```

FIGURE 7 – Code « objet 3d » du cube.

- la fonction de définition des points `set_C_points` reprend les appels de `set_point_` mais en les remplaçant par des appels à `set_point` ; l'appel `set_point(1)(0,0,0)` signifie que l'on définit le point 1 de cet objet là ; nous avons donc maintenant une notion *locale* de point ;
- enfin, une fonction de tracé `draw_C` qui indique comment l'objet doit être tracé.

L'instanciation proprement dite, c'est-à-dire l'opération qui va associer un objet à une classe, se fait avec `assign_obj("cube","C")`. Cette dernière opération définit l'objet *cube* comme une instance de la classe *C*. Elle conduit entre autres à l'appel de la fonction `def_C`, donc au calcul des points du cube. Il faut noter que dans cet exemple, les points du cube sont placés *avant* l'observateur.

Dans des dessins plus complexes (c'est par exemple le cas dans la figure 19), il est quelquefois nécessaire de faire dépendre certains points d'un objet de la position de l'observateur. Dans ce cas, outre l'appel à la fonction `assign_obj` (qui ne doit apparaître qu'une fois par objet), les positions de l'objet peuvent être recalculées avec `reset_obj` (`set_C_points` ne peut pas être utilisé directement car cette fonction ne dit pas comment calculer les numéros absolus des points).

La boucle principale du programme est désormais presque vide. Les définitions concernant l'observateur n'ont pas subi de modifications, si ce n'est celle concernant le point de visée. La fonction `point_of_view_obj` est maintenant utilisée pour viser non pas un point absolu, mais un point d'un objet, en l'occurrence le point 8 du cube.

Enfin, le cube est tracé avec `draw_obj`. Cette commande va à la fois faire la projection et appeler la commande `draw_C` (avec le paramètre `"cube"`) et il ne suffit donc pas d'appeler `draw_C("cube")`. La projection impose une corrélation entre les points locaux d'un objet et ceux du plan. Il n'est plus possible d'associer automatiquement le point 7 d'un objet à z_2 par exemple.

Par la suite, nous encapsulerons toujours nos constructions dans des classes, même si comme ici nous n'instancions la classe qu'une fois. Le tableau ci-dessous récapitule les principales commandes agissant sur les objets.

Définition de la classe C	`def_C` et `set_C_points`
Définition du tracé de la classe C	`draw_C`
Instanciation d'un objet	`assign_obj`
Opérations	`translate_obj`
	`rotate_obj`
	`scale_obj`
	`reset_obj`
Dessin d'un objet	`draw_obj`

4.3. La perspective

Par défaut, toutes les représentations se font en perspective linéaire (ou centrale), c'est-à-dire celle qui correspond à ce que voit un observateur ponctuel lorsque son champ est projeté sur un plan perpendiculaire à la direction de visée [9]. Les règles de construction légitime d'un dessin en perspective linéaire ont été codifiées par les peintres et architectes du Quattrocento. Cette perspective n'est pas très apparente sur la figure 3, car l'observateur est très éloigné du cube (nous sommes en principe à plus de deux mètres d'un cube de 10 cm

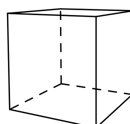


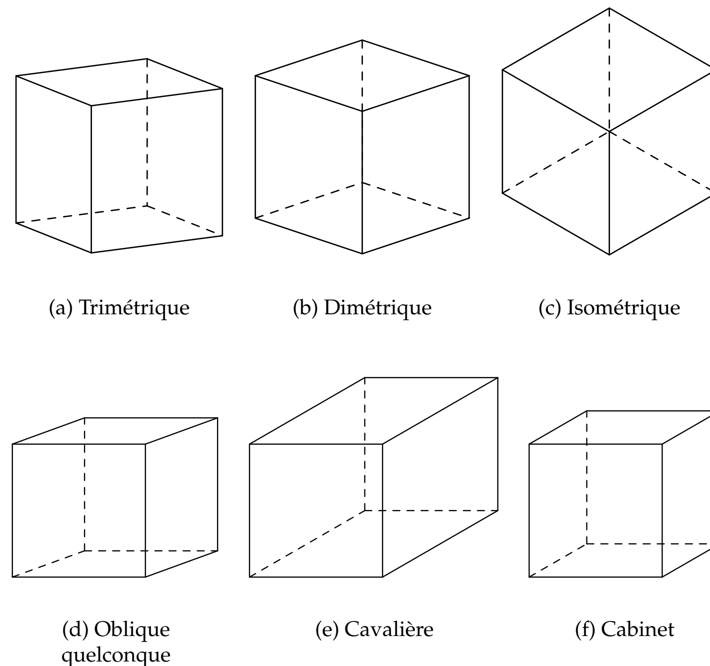
FIGURE 8 – Le cube avec un observateur cinq fois plus près que dans la figure 3.

de côté). En nous rapprochant du cube (et en diminuant `drawing_scale` pour que le dessin ne devienne pas trop grand), nous obtenons (sans avoir touché au cube) la figure 8. La perspective linéaire fait bien apparaître les lignes de fuite. (Notons que nous avons mis dans cet exemple les arêtes cachées en pointillés. Le cube étant défini comme il l'est dans la figure 7, il n'est pas possible de déterminer automatiquement ce qui est visible et ce qui ne l'est pas, puisque nous n'avons rien dit sur les faces. Mais si le cube est défini comme un polyèdre avec l'extension `3dpoly` (cf. [11]), l'élimination pourra se faire automatiquement. Toutefois, cette élimination n'est pour l'instant implantée que pour des objets convexes isolés. Dans le présent article, tous les pointillés sont introduits manuellement.)

Le *package* `3d` offre d'autres possibilités de perspectives en modifiant la valeur de `projection_type`. Une valeur de 0 correspond à la perspective linéaire. Une valeur de 1 correspond à une perspective parallèle, où toutes les projections se font parallèlement à la direction de visée et perpendiculairement au plan de projection. Cette perspective est différente de la perspective cavalière. Elle correspond à un observateur à l'infini, mais qui regarde la scène avec un télescope de puissance infinie. Comme cette perspective ne modifie pas les tailles, il est en général nécessaire de réduire la taille de la projection en modifiant `drawing_scale`.

Une première catégorie de projections parallèles sont les projections isométrique (dite aussi perspective militaire), dimétrique et trimétrique, regroupées usuellement sous le nom de perspectives axonométriques. Toutefois, selon Krikke [8], ces projections sont dénommées ainsi à tort. Alors que la perspective isométrique a été inventée par William Farish en 1822 pour répondre à des besoins créés par la révolution industrielle, la vraie perspective axonométrique serait une perspective ayant son origine en Chine et au Japon, en particulier parce qu'elle se prête bien à une présentation par rouleaux. Dans toutes les perspectives parallèles, l'apparence d'un objet ne dépend que de son orientation, pas de sa distance. Un objet éloigné n'apparaît pas plus petit qu'un objet rapproché.

La valeur 2 de `projection_type` correspond aux perspectives obliques qui sont des perspectives parallèles, mais où le plan de projection n'est pas nécessairement perpendiculaire à la direction de projection. En général, le



(a) Trimétrique (b) Dimétrique (c) Isométrique

(d) Oblique quelconque (e) Cavalière (f) Cabinet

FIGURE 9 – Perspectives ou projections parallèles. Les projections du haut sont des projections où le plan de projection est perpendiculaire à la direction de projection. Ces projections sont aussi appelées axonométriques par certains auteurs [3]. Les projections du bas sont des projections obliques, où le plan de projection n'est pas perpendiculaire à la direction de projection. La projection trimétrique est le cas quelconque des projections axonométriques. Dans la projection dimétrique, deux des axes sont à la même échelle et dans la projection isométrique, les trois axes sont représentés à la même échelle. Dans les projections obliques, l'une des faces de l'objet est en général parallèle au plan de projection et apparaît ainsi sans déformation. Dans les trois cas représentés, l'une des facettes du cube projeté est bien un carré. L'angle des lignes fuyantes est variable. Dans la projection cavalière, les lignes fuyantes sont représentées à la même échelle que les lignes dans le plan de projection, ce qui donne un aspect non naturel à cette projection (même si c'est une authentique projection oblique sans aucune déformation). Dans la projection cabinet, les lignes fuyantes sont représentées à l'échelle $1/2$ ce qui donne une représentation plus naturelle, bien qu'il n'y ait pas de point de fuite. Certaines projections obliques sont aussi appelées planométriques, par exemple quand la face parallèle au plan de projection représente un plan au sol et lorsque les lignes verticales apparaissent verticales en projection.

plan de projection est pris parallèle à l'une des faces de l'objet. Les perspectives obliques les plus courantes sont la perspective cavalière et la perspective cabinet. L'*axonométrie asiatique*, où un axe horizontal est perpendiculaire à la direction de visée, semble aussi être une perspective oblique.

Le *package* `3d` permet d'obtenir l'une quelconque de ces perspectives. Dans le cas des perspectives parallèles, la position de l'observateur ne sert qu'à déterminer sa direction de regard, pas le degré de proximité. Dans le cas des projections obliques, le plan de projection étant désolidarisé de l'observateur, il est nécessaire de le fixer explicitement.

La figure 9 récapitule les différentes perspectives parallèles.

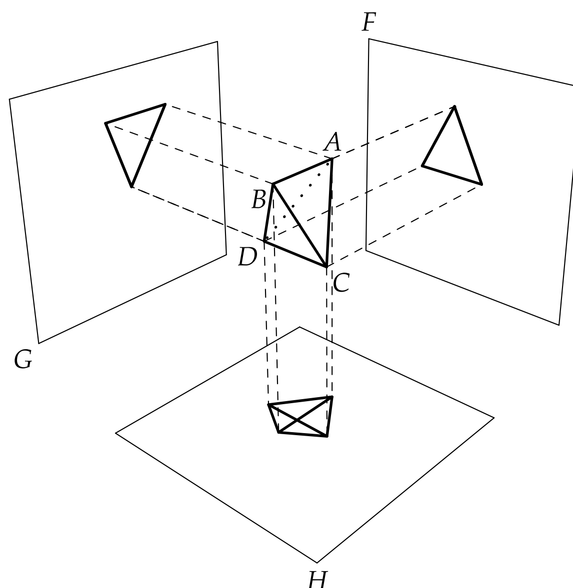


FIGURE 10 – Projections orthogonales.

Un autre type de représentation utilise des projections orthogonales (ou orthographiques) complémentaires. Il s'agit de décrire un objet par plusieurs projections orthogonales sur des plans eux-mêmes orthogonaux (voir figure 10). L'emploi de plusieurs vues orthogonales et leur croisement sont connus depuis l'Antiquité, mais ont été exposés géométriquement pour la première fois par Piero della Francesca à la Renaissance (cf. [9, p. 70]). Les croisements de vues orthogonales peuvent être utilisés pour construire une vue en perspective linéaire. Par exemple, dans la figure 10, si les deux projections sur les plans F et G sont représentées dans le même plan, la projection centrale sur le plan

H peut être déterminée par recoupements. Albrecht Dürer s'est par exemple servi de cette technique pour construire l'intersection d'un cône et d'un plan dans une gravure célèbre [1].

Par la suite, Gaspard Monge a étudié systématiquement la méthode de double projection et l'a codifiée dans sa *Géométrie descriptive* (1799). Sa méthode de construction est quelquefois appelée construction ou projection de Monge. Les développements de cette technique ont abouti aux représentations standard en dessin technique, où l'on représente une vue de face d'un objet, à sa gauche la vue de droite, à sa droite la vue de gauche, au-dessous la vue de dessus, etc. (Notons que cette représentation n'est pas complètement standard puisqu'aux États-Unis, la vue de gauche est à gauche, la vue de droite à droite, etc.) Le *package* `3d` ne fournit pour l'instant pas de moyen automatique de représentation de ces perspectives, mais elle peuvent être simulées sans difficultés.

4.4. Indices locaux et absolus des points

Nous avons vu que chaque point ou chaque vecteur défini avec `new_point` ou `new_vec` correspond à un élément d'une pile. Un point est alors donné par son indice dans cette pile. L'indice local d'un point est alors le numéro de ce point tel qu'il apparaît dans la définition de l'objet auquel il appartient, s'il a été défini comme l'un des points constituant un objet. Lorsque nous avons défini le point 2 de notre cube avec `set_point`, 2 n'était pas l'indice de ce point dans la pile, mais un indice relatif au début de cet objet dans la pile.

Dans certains cas, il est nécessaire de connaître l'indice absolu d'un point, par exemple parce que certaines fonctions en ont besoin. Cet indice absolu est obtenu par la fonction `pnt`. Par exemple, pour calculer le milieu des deux points 1 et 2 et mettre sa valeur dans le point 3, tous ces points étant des points locaux d'un objet, une solution est de faire :

```
vec_sum_(pnt(3),pnt(1),pnt(2));
vec_mult_(pnt(3),pnt(3),.5);
```

`vec_sum_` (somme de deux vecteurs) et `vec_mult_` (multiplication d'un vecteur par un scalaire) (cf. figure 11) prennent en effet pour paramètre des indices absolus. Il ne suffit pas de créer des variantes de ces fonctions pour le cas où les points (vecteurs) sont locaux (ces variantes existent et s'appellent ici `vec_sum` et `vec_mult`) car il y a souvent des cas intermédiaires faisant intervenir des points locaux d'un objet, voire de plusieurs objets, et des points donnés par leurs indices absolus (comme `Obs` par exemple). Par conséquent,

FIGURE 11 – Deux opérations sur les vecteurs.

nous avons fourni des variantes des fonctions pour les plus courantes d'entre elles, mais pas pour toutes, et pas toutes les variantes imaginables. Il faut dans certains cas tout de même recourir à la fonction `pnt`. Toutefois, pour l'exemple précédent, la fonction `mid_point` peut être avantageusement utilisée :

```
mid_point(3,1,2);
```

La variante non locale, ou absolue, de `mid_point` est `mid_point_`. Les fonctions plus « internes » du *package* `3d` sont affublées de ce « _ » terminal. Ainsi, `mid_point(3,1,2)` est équivalent à `mid_point_(pnt(3),pnt(1),pnt(2))`.

Si nous avions voulu définir un point non local p (défini en dehors d'un objet) comme étant le milieu de deux points locaux, nous aurions écrit :

```
mid_point_(p,pnt(1),pnt(2));
```

Dans tous les cas, il faut prendre garde à n'utiliser `pnt` qu'au sein d'un objet et plus précisément uniquement dans les fonctions `def` et `draw` de cet objet.

4.5. Structures de l'espace

Les objets définissables avec le *package* `3d` sont des objets en général assez complexes qui vont finir par être projetés et dessinés. Ces objets ne sont pas particulièrement adaptés à un traitement mathématique. Or, dans les constructions géométriques, que ce soit dans le plan ou l'espace, interviennent des concepts très simples comme les droites, les plans, et d'autres surfaces ou volumes définis mathématiquement. Ces concepts, que nous appelons ici structures, servent souvent d'intermédiaire pour trouver de nouveaux points ou de nouvelles courbes. Les structures sont rarement tracées. Nous ne tracerons

jamais une droite, mais uniquement un segment de cette droite. Nous ne tracerons jamais un plan, mais uniquement par exemple un rectangle de ce plan ou simplement quelques points de ce plan.

Pour faciliter la manipulation de ces structures, nous les avons créées d'une manière différente et plus simple que les objets. Ces structures ont quelques analogies avec les types primitifs du langage Java. Chaque structure pourrait être enveloppée par un objet ou associée à un objet, mais nous ne le faisons pas ici.

Les structures sont définies dans le module `3dgeom`. Elles doivent elles aussi être allouées et libérées.

4.5.1. Droites

La structure la plus simple que nous définissons est la droite. Une droite est définie à partir de deux points. Par exemple, pour définir la droite l passant par les points locaux 4 et 6, on écrit :

```
new_line(l)(4,6);
```

Cette fonction (d'analogue absolu `new_line_`) mémorise les deux points, de telle sorte qu'une modification ultérieure des points ne modifie pas la droite. La ligne n'est donc qu'initialement attachée aux points. Toutefois, cela est rarement une gêne car les structures sont souvent introduites localement pour faire une construction. De plus, il est possible de créer une version de la fonction `new_line` qui ne duplique pas les points.

Il arrive que l'on veuille définir une droite, mais que les points ne soient pas encore calculés, ou plus couramment, que l'on veuille redéfinir une droite à partir d'un autre couple de points, pour commencer une autre construction. La commande `set_line` (ou `set_line_`) peut alors être utilisée :

```
set_line(l)(4,8);
```

Enfin, lorsque la droite n'est plus nécessaire, elle est libérée avec `free_line` (qui n'a qu'une version) :

```
free_line(l);
```

Notons que les structures définies dans un objet doivent toujours être libérées dans cet objet et, comme pour les points, dans l'ordre inverse de leur allocation.

4.5.2. *Plans*

Un plan est défini de manière analogue à une droite, mais à partir de trois points :

```
new_plane(p)(i,j,k);
set_plane(p)(i,j,k);
free_plane(p);
```

Les analogues absolus sont `new_plane_` et `set_plane_`.

4.5.3. *Autres structures*

D'autres structures (planes ou non, comme les cercles, sphères, etc.) sont définies dans `3dgeom`, mais elles n'ont pas encore toutes été développées. Il est très facile d'ajouter de nouvelles structures et des fonctions qui les manipulent.

4.6. Constructions élémentaires

4.6.1. *Définitions de plans*

L'emploi des structures permet de simplifier considérablement les constructions géométriques dans l'espace. Par exemple, pour dessiner les projections des sommets du tétraèdre dans la figure 10, nous avons défini les trois plans de projection avec `new_plane` :

```
new_plane(f)(9,10,11);
new_plane(g)(13,14,15);
new_plane(h)(5,6,7);
```

4.6.2. *Verticales d'un plan*

Nous avons ensuite abaissé les verticales des sommets sur tous ces plans, au moyen de la fonction `def_vert_pl` :

```
def_vert_pl(17)(1)(h);
def_vert_pl(18)(2)(h);
def_vert_pl(19)(3)(h);
def_vert_pl(20)(4)(h);
...
```

Cette fonction prend un point et un plan et détermine le pied de la verticale au plan passant par le point donné en paramètre (ici, le second).

4.6.3. *Intersections entre droites et plans*

L'une des fonctions de `3dgeom` permet de calculer l'intersection d'une droite et d'un plan :

```
boolean b;
b:=def_inter_p_l_pl(i)(l)(p);
```

L'intersection de la droite l et du plan p, si elle existe, est calculée. S'il y a une intersection qui se réduit à un point, la fonction renvoie `true`, sinon `false`. Le point rendu est i (indice local).

Nous allons illustrer cette fonction avec un énoncé de géométrie de lycée : $ABCD$ est un tétraèdre tel que $AB = 3$, $AC = 6$, $AD = 4.5$. I est le point de $[AB]$ tel que $AI = 1$ et J est le point de $[AC]$ tel que $AJ = 4$. Il s'agit de déterminer l'intersection de la droite (IJ) avec le plan (BCD). Commençons par construire le tétraèdre (figure 12).

Remarquons tout d'abord que plusieurs tétraèdres satisfont l'énoncé : B, C et D sont indépendants. Pour obtenir une construction assez générale, il faut la paramétrer. A peut par exemple être placé en $(0,0,0)$, B en $(3\cos\beta, 3\sin\beta, 0)$, C en $(6\cos\gamma, 6\sin\gamma, 0)$ et D obtenu à partir de deux rotations, l'une autour de $\vec{k}$, l'autre autour d'un vecteur orthogonal à $\vec{k}$. En METAPOST, cela donne :

```
set_point(1)(0,0,0); % A
set_point(2)(3*cosd(b),3*sind(b),0); % B
set_point(3)(6*cosd(c),6*sind(c),0); % C
new_vec(v_a); new_vec(v_b);
vec_def_vec_(v_a,vec_I);  % $\vec{v}_a \leftarrow \vec{\imath}$
vec_rotate_(v_a,vec_K,d); % rot. de $\vec{v}_a$ d'un angle $d$ autour de $\vec{k}$
vec_prod_(v_b,v_a,vec_K); % $\vec{v}_b \leftarrow \vec{v}_a \wedge \vec{k}$
vec_rotate_(v_a,v_b,e);   % rot. de $\vec{v}_a$ d'un angle $e$ autour de $\vec{v}_b$
vec_mult_(v_a,v_a,4.5);
vec_sum_(pnt(4),pnt(1),v_a); % D
free_vec(v_b); free_vec(v_a);
% Détermination de I et J :
% $I = A + \overrightarrow{AB}/\|\overrightarrow{AB}\|$
vec_diff(5,2,1); % $\overrightarrow{V_5} \leftarrow \overrightarrow{AB}$
vec_unit(5,5);   % $\overrightarrow{V_5} \leftarrow \overrightarrow{AB}/\|\overrightarrow{AB}\|$
vec_sum(5,5,1);  % $I \leftarrow A + \overrightarrow{AB}/\|\overrightarrow{AB}\|$
% $J = A + 4 \cdot \overrightarrow{AC}/\|\overrightarrow{AC}\|$
```

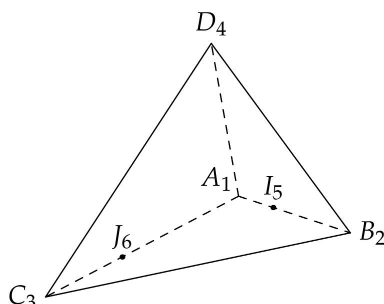


FIGURE 12 – Tétraèdre : première construction.

```
vec_diff(6,3,1); % $\overrightarrow{V_6} \leftarrow \overrightarrow{AC}$
vec_unit(6,6);   % $\overrightarrow{V_6} \leftarrow \overrightarrow{AC}/\|\overrightarrow{AC}\|$
vec_mult(6,6,4); % $\overrightarrow{V_6} \leftarrow 4\cdot\overrightarrow{AC}/\|\overrightarrow{AC}\|$
vec_sum(6,6,1);  % $J \leftarrow A+4\cdot\overrightarrow{AC}/\|\overrightarrow{AC}\|$
```

Sur la figure 12, les numéros des points ont été ajoutés en indice. Cette figure n'est évidemment pas bien adaptée à ce problème, car le plan (BCD) nous fait face. Nous allons donc déplacer l'observateur, par exemple en $C + 5\overrightarrow{DC} + 5\overrightarrow{BC} + 3\overrightarrow{CA}$ (figure 13). Pour effectuer le déplacement, nous avons accédé à des points du tétraèdre en dehors de celui-ci en utilisant la fonction `pnt_obj` (cette fonction permet aussi de définir des points d'un objet à partir de ceux d'un autre objet) :

```
new_vec(v_a); new_vec(v_b); new_vec(v_c);
vec_diff_(v_a,pnt_obj("tetra",3),pnt_obj("tetra",4)); % $\overrightarrow{DC}$
vec_mult_(v_a,v_a,5);                                 % $5\cdot\overrightarrow{DC}$
vec_diff_(v_b,pnt_obj("tetra",3),pnt_obj("tetra",2)); % $\overrightarrow{BC}$
vec_mult_(v_b,v_b,5);                                 % $5\cdot\overrightarrow{BC}$
vec_diff_(v_c,pnt_obj("tetra",1),pnt_obj("tetra",3)); % $\overrightarrow{CA}$
vec_mult_(v_c,v_c,3);                                 % $3\cdot\overrightarrow{CA}$
vec_sum_(Obs,pnt_obj("tetra",3),v_a); % $C+5\overrightarrow{DC}$
vec_sum_(Obs,Obs,v_b);                % $C+5\cdot\overrightarrow{DC}+5\cdot\overrightarrow{BC}$
vec_sum_(Obs,Obs,v_c);                % $C+5\cdot\overrightarrow{DC}+5\cdot\overrightarrow{BC}+3\cdot\overrightarrow{CA}$
free_vec(v_c); free_vec(v_b); free_vec(v_a);
```

Nous arrivons maintenant au calcul d'intersection de la droite (IJ) avec le plan (BCD) (figure 14).

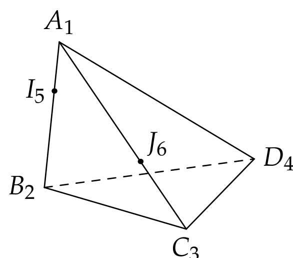


FIGURE 13 – Tétraèdre : seconde construction.

```
new_plane(bcd)(2,3,4);
new_line(ij)(5,6);
boolean b;
b:=def_inter_p_l_pl(7)(ij)(bcd):
if not b: message "pas d'intersection"; fi;
free_line(ij);
free_plane(bcd);
```

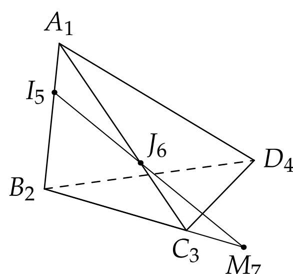


FIGURE 14 – Tétraèdre : troisième construction.

Deux autres intersections peuvent être calculées, à partir du point K milieu de $[AD]$ (figure 15). Les trois intersections sont alors alignées. (On voit qu'en fait L, M et N sont sur l'intersection des plans (IJK) et (BCD).) Nous avons indiqué l'alignement avec un segment dépassant légèrement à chaque extrémité en utilisant

```
draw_line_extra(9,10)(-0.1,1.1);
```

Le second couple de paramètres indique la valeur du dépassement. $(0, 1)$ correspond à un dépassement nul et une valeur inférieure (resp. supérieure) pour le premier (resp. second) paramètre produit un dépassement.

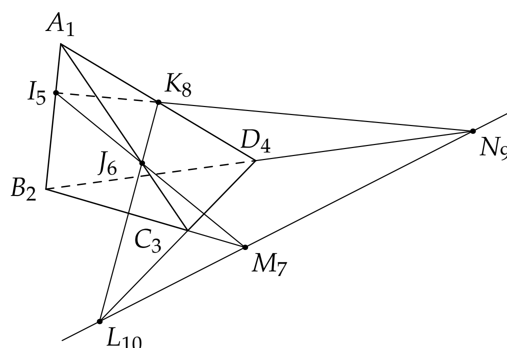


FIGURE 15 – Tétraèdre : quatrième construction.

La figure 15 illustre aussi le célèbre théorème de Girard Desargues (1639), qui est la pierre angulaire de la géométrie projective. En effet, d'après ce théorème, étant donné deux triangles BCD et IJK du plan, les droites (BI), (CJ), (DK) ont une intersection si et seulement si les intersections de (IK) et (BD), de (IJ) et (BC), et de (KJ) et (DC) sont alignées. Dans notre cas de figure, les droites (BI), (CJ), (DK) ont bien une intersection, à savoir A, et les points d'intersection L, M et N sont alignés. Le théorème dit que la réciproque est aussi vraie. Vu dans l'espace, le théorème est très simple, mais une preuve purement plane est difficile.

Une autre fonction de `3dgeom` permet de calculer directement l'intersection entre deux plans :

```
b:=def_inter_l_pl_pl(l)(p)(q);
```

où `B` est de type `boolean`. Si l'intersection entre les plans p et q est une droite, la fonction renvoie `true` et la droite est stockée dans l. Sinon, la fonction renvoie `false`. Voyons sur un exemple comment cette fonction peut s'utiliser. Considérons le tracé d'un tétraèdre $SABC$ dont on connaît les longueurs des arêtes SA, SB et SC, ainsi que les angles $\widehat{ASC}$, $\widehat{ASB}$ et $\widehat{BSC}$. La figure 16 montre un tel tétraèdre avec $SA = 9$, $SB = 8$, $SC = 4$, $\widehat{ASC} = 60°$, $\widehat{ASB} = 40°$ et $\widehat{BSC} = 30°$. Contrairement à l'exemple précédent, nous n'avons pas ici une grande marge de manœuvre pour placer les points. Nous pouvons bien sûr commencer à construire par exemple le triangle SAC. Il ne reste alors qu'à placer le sommet B.

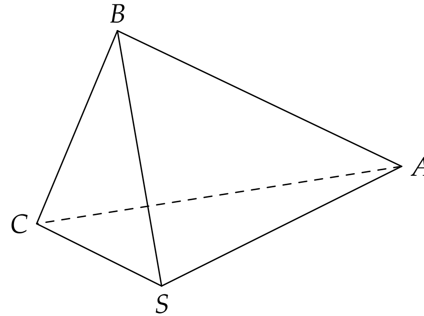


FIGURE 16 – Un tétraèdre spécifié par trois longueurs et trois angles.

Les angles $\widehat{ASB}$ et $\widehat{BSC}$ étant donnés, B se situe manifestement sur deux cônes : le cône d'axe (SA), de sommet S et d'angle $\widehat{ASB}$ et le cône d'axe (SC), de sommet S et d'angle $\widehat{CSB}$. Ces deux cônes ont en général pour intersection deux droites et B se situe sur l'une de ces intersections à la distance SB de S.

Avec une implantation convenable d'une structure de cône, l'intersection peut bien sûr être déterminée automatiquement. Mais il est aussi possible de procéder avec des moyens plus limités en constatant que nous pouvons abaisser les hauteurs $[SH]$ et $[SK]$ issues de B pour chacun des triangles SAB et SBC. Par exemple, $SH = SB \cdot \cos(\widehat{ASB})$. Nous pouvons ensuite définir les plans orthogonaux aux droites (SA) et (SC) passant par les deux pieds de hauteurs. La fonction

```
def_orth_pl_l_p(p)(l)(i);
```

construit le plan p orthogonal à la droite l et passant par le point local i.

L'intersection entre les deux plans ainsi construits peut ensuite être calculée. Cette intersection est une droite perpendiculaire au plan du triangle SAC. Sur cette droite, nous cherchons enfin un point B situé à une distance donnée de S. L'appel de la fonction

```
b:=def_point_at(i)(d)(j)(l);
```

où `b` est une variable de type `boolean`, définit le point local i comme étant un point de la droite l situé à la distance $|d|$ du point local j si ce point existe. Dans ce cas, la valeur de retour de la fonction est `true`, sinon `false`. En général,

deux points satisfont la condition et la fonction renverra l'un ou l'autre de ces points selon le signe de d.

L'essentiel de la figure est donc produit par les commandes :

```
new_point(h); new_point(k);
set_point(1)(0,0,0);   % S
set_point(2)(lsa,0,0); % A
set_point(4)(lsc*cosd(aasc),lsc*sind(aasc),0); % C
vec_diff_(h,pnt(2),pnt(1));
vec_unit_(h,h); vec_mult_(h,h,lsb*cosd(aasb));
vec_sum_(h,h,pnt(1));  % H
vec_diff_(k,pnt(4),pnt(1));
vec_unit_(k,k); vec_mult_(k,k,lsb*cosd(absc));
vec_sum_(k,k,pnt(1));  % K
new_plane(hp)(1,1,1);  % initialisation à trois points
new_plane(kp)(1,1,1);  % idem
new_line(sa)(1,2);     % (SA)
new_line(sc)(1,4);     % (SC)
new_line(inter)(1,1);  % ligne d'intersection des deux plans
def_orth_pl_l_p_(hp)(sa)(h); % plan orthogonal à (SA) en H
def_orth_pl_l_p_(kp)(sc)(k); % plan orthogonal à (SC) en K
if def_inter_l_pl_pl(inter)(hp)(kp): % il y a une intersection
  if not def_point_at(3)(-lsb,1)(inter): % B
    message "Ne devrait pas se produire";
  fi;
else:
  message "PROBLÈME (sans doute angle ASC trop petit)";
  set_point(3)(1,1,1);
fi;
free_line(inter); free_line(sc); free_line(sa);
free_plane(kp); free_plane(hp);
free_point(k); free_point(h);
```

L'ensemble de la construction est donné en figure 17. Pour la réaliser, nous avons « délocalisé » certains points comme les bases des hauteurs et le point d'intersection de la verticale en B avec le plan (SAC). Pour réaliser les angles droits, nous avons utilisé les commandes `def_right_angle` pour la définition et `draw_double_right_angle` pour le tracé. Chaque angle droit consiste en effet en deux segments, définis à partir de trois points. Ces trois points sont de nouveaux points de l'objet. Par exemple, l'un des angles droits est créé avec

```
def_right_angle(7,8,9,5,1,3);
```

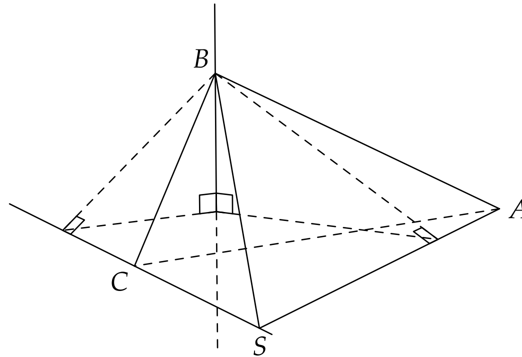


FIGURE 17 – Un tétraèdre spécifié par trois longueurs et trois angles (construction).

Cela signifie que trois points (numérotés localement 7, 8 et 9) sont introduits et qu'ils se placent selon l'angle déterminé par le triangle des points (5,1,3).

Le tracé, quant à lui, est plus simple :

```
draw_double_right_angle(7,8,9,5);
```

4.7. Compléments visuels

4.7.1. Représentations de plans

Un plan est souvent représenté à l'aide de quatre points particuliers formant un rectangle. Ces points doivent être définis. Le tracé d'un rectangle horizontal correspondant au plan (*SAC*) de la figure 17 peut se faire ainsi (cf. figure 18) :

```
set_point(14)(-2,-2,0); % p1
set_point(15)(11,-2,0); % p2
set_point(16)(11,10,0); % p3
set_point(17)(-2,10,0); % p4
```

Ces points sont ensuite reliés à l'aide de la commande `draw_lines`.

FIGURE 18 – L'ajout d'un plan au dessin de la figure 17.

4.7.2. *Parties cachées et intersections visuelles*

Comme le montre la figure 18, il n'y a pas automatiquement (pour l'instant) d'élimination des parties cachées ou un traitement spécial de celles-ci. Il est nécessaire de gérer soi-même les pointillés et ceci n'est pratique que dans le cas d'images fixes. Avec des animations, la perspective peut subir de telles variations qu'il est souhaitable d'avoir une résolution automatique de ce problème.

Le problème qui se pose dans l'élimination (ou le traitement spécial) des parties cachées est cependant assez simple à exprimer et à résoudre sur une vue fixe. Il s'agit finalement de déterminer des points d'intersection « apparents » entre deux courbes. Dans l'exemple précédent, certains côtés du tétraèdre semblent rencontrer des côtés du rectangle représentant le plan, bien qu'en réalité ils ne les rencontrent pas dans l'espace.

Si le plan devait être représenté d'une manière plus claire, il faudrait déterminer les intersections apparentes (ou visuelles) des côtés $[BA]$ et $[BC]$ avec le côté le plus éloigné du rectangle du plan. Comment s'y prendre ?

Une manière est de construire un point du segment $[p_2p_3]$, mais en fonction de la position de l'observateur. L'intersection entre les plans (Obs, p_2, p_3) et (Obs, A, B) est une droite passant par l'observateur et par le point de (p_2p_3) qui nous intéresse. Il suffit alors de déterminer l'intersection entre cette droite et la droite (p_2p_3). Cette intersection étant calculée dans l'espace, il faut prendre garde aux erreurs d'arrondis pouvant facilement empêcher la rencontre de deux droites qui pourtant devraient se croiser. La fonction `def_inter_p_l_l` permet de déterminer le milieu des deux points de chaque droite où celle-ci est la plus proche de l'autre. Cette fonction renvoie aussi la

distance entre les deux points. Ceci permet de déterminer un point de (p_2p_3) correspondant à l'interruption visuelle causée par le segment $[BA]$. De même, il est possible de déterminer le point correspondant à l'interruption visuelle causée par le segment $[BC]$. Ces deux points, avec p_2 et p_3, permettent alors de tracer un plan plus naturel. C'est ce qui est fait dans la figure 19.

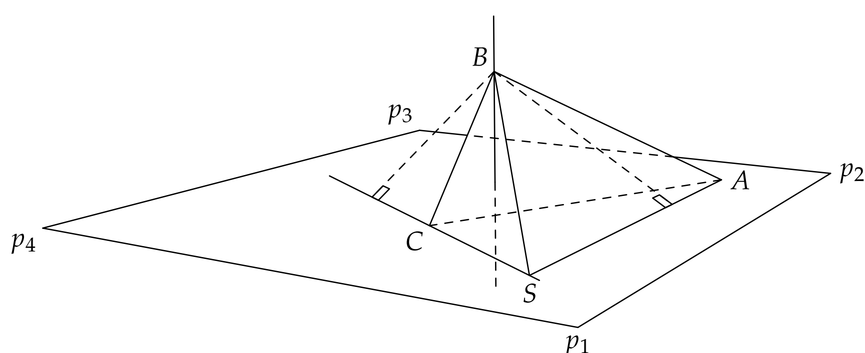


FIGURE 19 – L'interruption d'un plan.

Cette procédure est accessible avec la fonction `def_visual_inter` qui prend quatre points locaux et en calcule un cinquième :

```
boolean b;
b:=def_visual_inter(i)(j,k,l,m);
```

Si la fonction renvoie `true`, le point i se situe sur la droite (jk) à l'intersection apparente des droites (jk) et (lm).

Il faut noter que dans une perspective centrale les intersections calculées dépendent de la position de l'observateur. Si celle-ci change, il faut les recalculer. Par conséquent, comme nous l'avons déjà indiqué, il est nécessaire de faire un appel à la fonction `reset_obj` après la redéfinition de la position de l'observateur.

Dans une projection parallèle, l'observateur n'intervient plus dans le calcul des intersections visuelles, mais l'interface reste identique. Le principe de calcul des intersections est presque le même, si ce n'est que les plans dont on calcule l'intersection sont déterminés non pas par l'observateur et un segment, mais par la direction de projection et un segment.

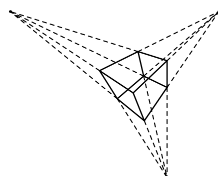


FIGURE 20 – Les trois points de fuite classiques d'un cube.

4.7.3. *Points de fuite*

La représentation d'un objet comportant des segments parallèles par une perspective centrale fait apparaître des lignes et points de fuite. Les droites projetées ne sont en général plus parallèles et se recoupent. Un exemple de représentation de cube avec trois points de fuite est donné en figure 20. Les représentations classiques distinguent souvent les dessins à un, deux ou trois points de fuite. Les cas de dessins à un ou deux points de fuite sont des cas particuliers correspondant à des droites parallèles au plan de projection. Dans la figure 20, aucun des côtés du cube n'est parallèle au plan de projection, ce qui entraîne l'existence de points de fuite. Si le plan de projection avait été parallèle aux segments verticaux du cube, ces segments n'auraient pas eu de point de fuite. Si c'est toute une face du cube qui est parallèle au plan de projection, il n'y a plus qu'un seul point de fuite.

Les points de fuite correspondent à des points situés à l'infini dans l'espace, selon une direction passant par l'observateur et dirigée par un vecteur correspondant au segment de l'objet. Très souvent, certains points de fuite seront assez éloignés sur le dessin, voire en-dehors de celui-ci.

Les représentations classiques en architecture ou en peinture placent deux points de fuite sur une horizontale et un troisième point de fuite qui correspond aux lignes de fuite verticales. La figure 20 diffère de cette représentation parce que l'observateur n'est pas orienté selon un axe vertical.

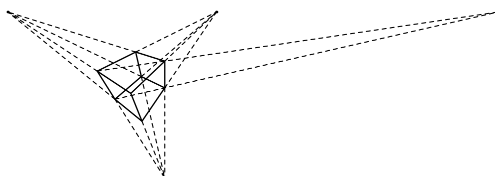


FIGURE 21 – Un quatrième point de fuite.

Le nombre de points de fuite différents ne correspond pas à des projections différentes, mais d'une part à des objets de l'espace positionnés différemment par rapport au plan de projection, d'autre part à des objets de nature différente. Une sphère n'aura évidemment pas de point de fuite ! Le nombre de points de fuite peut en fait être quelconque, y compris supérieur à trois. Il suffit de prendre un couple quelconque de droite parallèles de l'objet représenté. La figure 21 montre qu'outre les trois points de fuite classiques, on peut considérer les six points de fuite des segments diagonaux du cube dont l'un est représenté ici. (Notons que les points de fuite étant très sensibles à la position de l'observateur, il est assez difficile de trouver par tâtonnements un endroit où les neuf points de fuite du cube sont visibles simultanément dans un espace restreint.) Des objets plus complexes auraient encore davantage de points de fuite.

La détermination d'un point de fuite peut se faire simplement en calculant l'intersection entre le plan de projection et la droite passant par l'observateur et orientée par un vecteur de l'objet. Les lignes suivantes permettent de trouver le point de fuite du segment reliant deux points numérotés 1 et 5 :

```
def_screen_pl(screen); % définit le plan de projection
new_line(l)(1,5);
if not def_vanishing_point_p_l_pl(11)(l)(screen):
  message "pas de point de fuite";
  set_point(11)(0,0,0);
fi;
...
```

Tout comme les intersections visuelles, les points de fuite dépendent de la position de l'observateur et à chaque fois que le point de vue change (soit parce que l'observateur se déplace, soit parce que l'objet se déplace), il faut recalculer les points de l'objet avec `reset_obj`.

Il serait aussi possible de calculer les points de fuite directement dans le plan. Ce serait une simple application de `whatever` (cf. section 2).

4.7.4. *Ombrages*

Les ombrages correspondant à des projections sont simulés en grisant la partie projetée. Un exemple est donné figure 22. Dans cet exemple, un triangle est projeté sur un plan. Nous avons simplement calculé les projections des trois sommets du triangle et nous avons ensuite grisé la projection. Cette technique fonctionne quelle que soit la projection, tant que celle-ci se fait selon une droite et sur un plan ou un ensemble de plans.

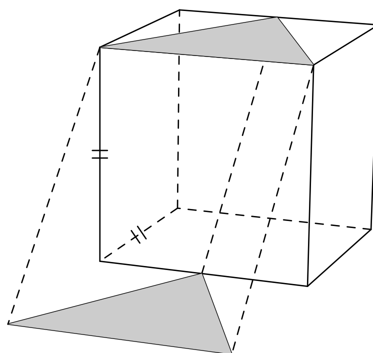


FIGURE 22 – Un cube avec un ombrage.

5. Modifications par rapport à la première distribution

Lors de la réalisation de la nouvelle version du *package* `3d`, nous y avons apporté diverses modifications qui nous semblaient aller dans le sens d'une plus grande homogénéité. Un certain nombre de fonctions élémentaires ont été renommées afin de mieux adhérer aux principes de nommage de fonctions que nous avons élaborés. Ainsi, notre premier article [11] sur le sujet n'est maintenant plus rigoureusement correct car les fonctions comme `vect_sum` doivent y être remplacées par `vec_sum_`. Tous les « `vect` » ont été remplacés par « `vec` ». Enfin, la création des vecteurs et points a légèrement changé. Les différences ne sont pas plus importantes, mais pour faciliter la transition au lecteur, nous avons fourni sur CTAN une version corrigée de l'article.

6. Conclusion et limites

Ce *package* comporte de nombreuses possibilités qui n'ont pas été évoquées et il est facile de lui en adjoindre de nouvelles, en particulier pour la manipulation de nouvelles structures. Nous avons passé sous silence l'affichage de courbes comme les cercles qui font intervenir un autre module qui fera l'objet d'une description ailleurs. Une documentation plus complète est livrée avec le *package*.

Une lecture attentive de cet article révèle aussi que les coordonnées d'un point n'ont été utilisées explicitement que pour quelques points construits directe-

ment avec `set_point_` ou `set_point`. Tous les autres points ont été obtenus par diverses opérations géométriques. Mais cela ne veut pas dire que les coordonnées ne sont pas accessibles ! Elles sont données par les fonctions `xval`, `yval` et `zval` appliquées à une référence de vecteur ou de point.

Ce *package* n'est bien sûr pas parfait et a des limites. Outre les limites de METAPOST, en particulier en capacités numériques (les dimensions sont limitées à 4096 points PostScript), le *package* que nous fournissons manque d'automatisme. Il faut encore beaucoup d'interventions manuelles. D'autres limitations concernent l'absence (pour l'instant) de traitement convenable et automatique de l'élimination des parties cachées. Enfin, la gestion des erreurs METAPOST ne sera pas simple pour qui n'est pas un peu expérimenté dans ce langage.

L'on ne manquera pas bien sûr de comparer ce travail à d'autres travaux allant dans le même sens. Tout d'abord, il est clair que nous ne prétendons pas rivaliser avec les logiciels professionnels de CAO ou de calcul formel. Nous voulons avant tout fournir un système léger et puissant d'aide à la construction géométrique, en particulier adapté à un cours de géométrie de lycée. Dans le monde TEX, il n'y a à notre connaissance que peu de travaux d'intégration de l'espace. METAGRAF (`http://w3.mecanica.upm.es/metapost/metagraf.php`), aussi basé sur METAPOST, est un système interactif avec une notion d'espace, mais qui ne semble pas fournir de possibilités de constructions géométriques, d'animations, de changements de perspective, etc. Le système PSTricks comporte un module 3D, mais celui-ci est peu développé. Les calculs sont faits par TEX et l'extension du système est fastidieuse. En dehors du monde TEX, divers langages 3D sont disponibles, en particulier OpenGL, qui va beaucoup plus loin que notre système. En ce qui concerne les systèmes d'enseignement de la géométrie, il faut citer en particulier le logiciel *Cabri-Géomètre* (cf. `http://www.cabri.net`).

7. Remerciements

Je tiens à remercier Nicolas Kisselhoff qui m'a conduit à développer le module `3dgeom` en me fournissant quelques figures de son cours de géométrie, Sami Alex Zaimi qui a développé la notion d'objet et a indirectement influencé ce travail et Pablo Argon qui m'a jadis convaincu de l'intérêt de METAPOST pour la géométrie de la règle et du compas. Hans Hagen a relu cet article et m'a poussé à le clarifier encore davantage, en particulier par l'introduction de nouvelles figures. Enfin, Jean-Michel Hufflen et Damien Wyart ont apporté des corrections et suggéré diverses améliorations.

Bibliographie

[1] Albrecht DÜRER, *Underweysung der messung / mit dem zirckel und richtscheyt / in Linien ebnen unnd gantzen corporen / durch Albrecht Dürer zu samen getzogen / und zu nutz aller kunstliebhabenden mit zu gehörigen figuren in truck gebracht / im jar. M.D.X.X.V.* (1525), traduit et présenté par Jeanne Peiffer dans Albrecht Dürer, *Géométrie*, collection Sources du savoir, Paris : Seuil, 1995.

[2] Michel GOOSSENS, Sebastian RAHTZ & Frank MITTELBACH, *The LaTeX Graphics Companion* (Reading, MA, USA : Addison-Wesley, 1997).

[3] Jean-Paul GOURRET, *Modélisation d'images fixes et animées* (Paris : Masson, 1994).

[4] Hans HAGEN, *MetaFun* (2000). `http://www.pragma-ade.com`

[5] John D. HOBBY, «A User's Manual for MetaPost», rapport technique 162, AT&T Bell Laboratories, Murray Hill, New Jersey (1992). `http://cm.bell-labs.com/who/hobby/MetaPost.html`

[6] Alan HOENIG, *TeX unbound. LaTeX & TeX Strategies for Fonts, Graphics, & More* (Oxford, New York : Oxford University Press, 1998).

[7] Donald E. KNUTH, *Computers & Typesetting, volume C: The METAFONTbook* (Reading, MA : Addison-Wesley Publishing Company, 1986).

[8] Jan KRIKKE, «Axonometry: A Matter of Perspective», IEEE Computer Graphics and Applications, vol. 20 (4), (2000) p. 7–11. `http://www.computer.org/cga/cg2000/pdf/g4007.pdf`

[9] Jean-Pierre LE GOFF, «De la perspective à l'infini géométrique», Pour la Science, (278), (2000) p. 66–72.

[10] Bertrand MEYER, *Conception et programmation orientées objet* (Paris : Eyrolles, 2000).

[11] Denis ROEGEL, «Creating 3D animations with METAPOST», TUGboat, vol. 18 (4), (1997) p. 274–283, `ftp://ftp.loria.fr/pub/ctan/graphics/metapost/macros/3d/tugboat/tb57roeg.pdf`, une version mise à jour se trouve au même endroit.