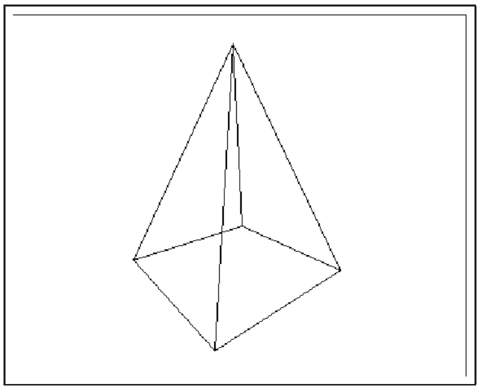
TP n°4 – L'espace en perspective

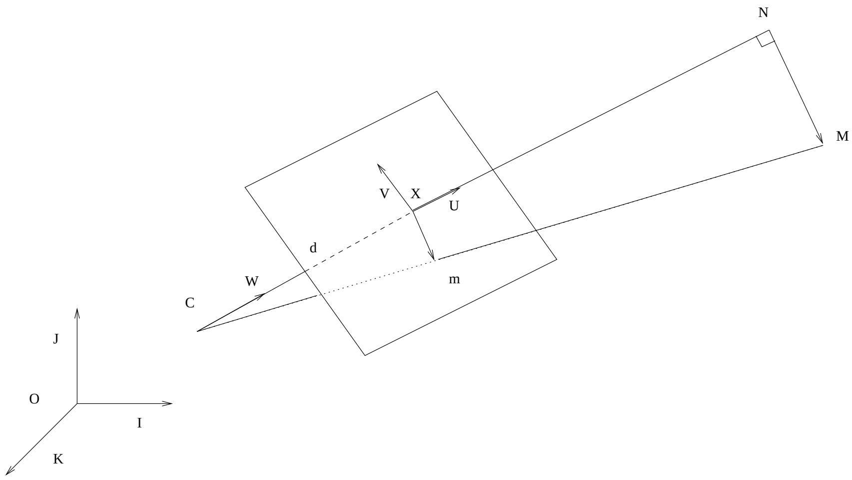
Projet de programmation (IK2)

L'objectif de ce TP est de pouvoir dessiner sur un écran à deux dimensions une figure définie dans un espace à trois dimensions. Cependant on se restreindra pour ce TP à afficher les figures sous la forme de « fil-de-fer » (ou *wireframe* en anglais). Voici par exemple ce que vous pourrez obtenir à la question 12 de ce TP :



Les figures seront définies par l'ensemble des segments qui la composent : la pyramide ci-dessus est décrite par 8 segments. Pour calculer cette image vous utiliserez la technique de projection conique[1] décrite ci-dessous.

Principe de la projection conique



Le dispositif de visualisation sera composé :

- d'un écran de projection de la taille de la fenêtre de visualisation ;
- d'une caméra représentée par un point, noté C et appelé centre de vision. Elle est orientée vers le centre de l'écran, noté X.

[1] Le mot *conique* se rapport au cône de vision issu de l'oeil, dans lequel on inscrit la zone d'affichage.

On appellera « axe de vision » la droite passant par C et X et on notera $\vec{w}$ le vecteur de norme 1 ayant la même direction que l'axe de vision. L'écran sur lequel on va projeter les figures sera dans un plan orthogonal à l'axe de vision. On notera $(X, \vec{u}, \vec{v})$ un repère orthonormé de ce plan de l'écran. On notera d la distance de C à X tel que : $\overrightarrow{CX} = d.\vec{w}$.

Pour tracer à l'écran un segment $[MM']$ d'une figure, il va falloir calculer les points m et m' – projections respectives sur le plan de l'écran des points M et M' – puis tracer la ligne $[mm']$ dans la fenêtre de visualisation.

La projection m sur l'écran d'un point M de l'espace sera l'intersection de la droite (CM) avec le plan $(X, \vec{u}, \vec{v})$. En nommant N le projeté orthogonal de M sur l'axe de vision, on pourra calculer les coordonnées du point m dans le repère $(X, \vec{u}, \vec{v})$ en utilisant la formule suivante :

$$\begin{aligned}
\overrightarrow{Xm} &= \frac{||\overrightarrow{Xm}||}{||\overrightarrow{NM}||}\overrightarrow{NM} && \text{(Vecteurs colinéaires)} \\
&= \frac{d}{CN}\overrightarrow{NM} && \text{(Théorème de Thalès)} \\
&= \frac{d}{CN}\Big(\big(\overrightarrow{NM}.\vec{u}\big)\vec{u} + \big(\overrightarrow{NM}.\vec{v}\big)\vec{v}\Big) && \text{(Par projection)}
\end{aligned}$$

Rappel géométrique — *Le produit scalaire x de deux vecteurs $\vec{u}$ et $\vec{v}$, noté $x = \vec{u}.\vec{v}$ permet notamment de calculer la longueur de la projection orthogonale de $\vec{v}$ sur $\vec{u}$ avec la formule :*

$$||\vec{v}_{\vec{u}}|| = \frac{\vec{u}.\vec{v}}{||\vec{u}||}$$

Il se calcule ainsi :

$$\begin{pmatrix} x_1 \\ y_2 \\ z_3 \end{pmatrix} . \begin{pmatrix} x_1 \\ y_2 \\ z_3 \end{pmatrix} = x_1.x_2 + y_1.y_2 + z_1.z_2$$

Rappel géométrique — *Le produit vectoriel $\vec{w}$ de deux vecteurs non-colinéaires $\vec{u}$ et $\vec{v}$, noté $\vec{w} = \vec{u} \wedge \vec{v}$ est un vecteur perpendiculaire au plan défini par $\vec{u}$ et $\vec{v}$ et de norme*

$$||\vec{w}|| = ||\vec{u}||.||\vec{v}||.\cos(\widehat{\vec{u}.\vec{v}})$$

Il se calcule ainsi :

$$\begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} \wedge \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} = \begin{pmatrix} y_1.z_2 - z_1.y_2 \\ z_1.x_2 - x_1.z_2 \\ x_1.y_2 - y_1.x_2 \end{pmatrix}$$

Description d'une figure

La première étape de ce programme est de définir les classes `Point`, `Segment` et `Figure` qui vont vous permettent de construire les figures à dessiner.

Rappel informatique — *Dans une méthode non-statique le mot-clé* `this` *sert à représenter l'objet courant, c'est-à-dire celui sur lequel la méthode est appelée.*

Rappel informatique — *Pour la classe* `Deug` *les couleurs sont représentées par un entier compris entre 0 et 255. L'entier 0 (resp. 255) correspond à la couleur noire (resp. blanche).*

Exercice 1 — *Classes de base* Créez les classes suivantes, en n'écrivant pour l'instant que les champs (avec l'attribut `private`), les accesseurs et les constructeurs :

Point Un `Point` est un point de l'espace, caractérisé par ses trois coordonnées réelles.

Segment Un `Segment` est défini par les deux `Points` représentant ses extrémités.

Figure Une `Figure` est constituée d'un ensemble de `Segments` et de la couleur qui sera utilisée pour dessiner cette figure. Vous utiliserez un tableau pour stocker l'ensemble des segments qui composent une figure.

Exercice 2 — *Ajout d'un segment à une figure* Ajoutez à la classe `Figure` une méthode qui permette d'ajouter un `Segment` à la `Figure` courante. Elle devra avoir la signature suivante :

```
void ajouteSegment(Segment s)
```

Indication — *Éventuellement cette méthode devra allouer un nouveau tableau de taille supérieur au tableau existant et y recopier les anciennes valeurs.*

Exercice 3 — *Une pyramide* Créez une classe `Exemple` qui va vous servir à tester votre programme au fur et à mesure. À l'aide des exercices 1 et 2, créez dans la méthode `main` de cette classe `Exemple` une figure représentant une pyramide à base carré de couleur noire.

Indication — *Vous pouvez utiliser les coordonnées suivantes :* $(-50, -50, 0)$, $(-50, 50, 0)$, $(50, 50, 0)$, $(50, -50, 0)$ *pour les coins du carré à la base de la pyramide et* $(0, 0, 150)$ *comme sommet de la pyramide.*

Vecteurs

La seconde étape est de définir la notion de vecteur. Ils seront utiles pour exprimer plus clairement les calculs de projection.

Exercice 4 — *Classe* `Vecteur` Créez une classe `Vecteur` caractérisée par ses trois coordonnées réelles (*vous pouvez reprendre la classe* `Vecteur` *du TP3 et la compléter*).

Pour faciliter l'écriture de calcul avec des vecteurs, on impose que les champs d'un vecteur ne soient pas modifiables : pour cela vous ajouterez le mot-clé `final` en plus de `private` devant la définition de chacun de ses champs.

Ajoutez un constructeur prenant les trois coordonnées réelles en argument.

Ajoutez les accesseurs nécessaires.

Ajoutez un constructeur prenant deux `Point`s en arguments et qui définit le `Vecteur` allant du premier `Point` au second.

Exercice 5 — *Opérations sur les vecteurs* Ajoutez à la classe `Vecteur` des méthodes pour les opérations suivantes.

- produit par un réel :
  `public Vecteur produit(double facteur)`
- produit scalaire :
  `public double scalaire(Vecteur v)`
- produit vectoriel :
  `public Vecteur vectoriel(Vecteur v)`
- normalisation (vecteur de même direction et de norme 1) :
  `public Vecteur normalise()`

Comme il a été interdit avec le mot-clé `final` de modifier les champs d'un vecteur, les méthodes `produit` et `normalise` doivent construire un nouveau vecteur.

Exercice 6 — *Test* Si vous ne l'avez pas déjà fait, ajoutez une méthode d'affichage à la classe `Vecteur` et testez l'ensemble des méthodes que vous avez définies dans la classe `Vecteur`.

Visualisation par projection

Maintenant que l'ensemble des outils nécessaires aux calculs de projections est défini, vous allez pouvoir définir la classe `Camera` pour effectuer ces calculs.

Exercice 7 — *Classe* Camera Créez la classe `Camera` avec les champs suivants :

```
private Point C;      // Position de la caméra
private Vecteur W;    // Direction du regard de la caméra
private double d;     // Distance de l'écran à la caméra

private Point X;      // Centre de l'écran
private Vecteur U,V; // De sorte que (X,U,V,W) soit
                      // un repère orthonormé de l'écran
private int l,h;      // Largeur et hauteur de l'écran
```

Comme vous l'avez peut-être remarqué certains de ces champs sont redondants. Par exemple le `Point X` peut-être calculé à partir du `Point C`, du `Vecteur W` et de la distance `d`. Écrivez une méthode `void update()` mettant à jour les champs `X`, `U` et `V` en fonction des autres champs, afin notamment que le constructeur de la classe `Camera` puisse être défini ainsi :

```
public Camera(Point C, Vecteur W, double d, int l, int h) {
  this.C = C;
  this.d = d;
  this.l = l;
  this.h = h;
  this.W = W.normalise();
  this.update();
}
```

Dans un premier temps[2] vous pouvez supposer que l'axe de vision est parallèle au sol. C'est-à dire que les vecteurs $\vec{u}$ et $\vec{w}$ appartiennent au plan $(O, \vec{i}, \vec{j})$ et que le vecteur $\vec{v}$ est égal au vecteur $\vec{k}$.

Exercice 8 — *Point du plan* Créez la classe `Point2D` caractérisée par ses deux coordonnées réelles afin de représenter les points du plan de l'écran. Ajoutez y les constructeurs et les accesseurs nécessaires.

Exercice 9 — *Projection* Ajoutez à la classe `Camera` une méthode de projection d'un point de l'espace vers un point de l'écran de la `Camera` courante, telle que le `Point X` soit représenté par le `Point2D` de coordonnées $(0, 0)$:

```
public Point2D projeter(Point P)
```

Indication — *Vous pouvez dans un premier temps calculer les coordonnés du point* `N` *puis utiliser la formule de projection proposée dans l'introduction.*

Exercice 10 — *Dessin* Ajoutez à la classe `Camera` une méthode de dessin d'une ligne en prenant soin de dessiner le point X au centre de la fenêtre et de ne pas inverser le haut et le bas de l'écran.

```
public void drawLine(Point2D p1, Point2D p2)
```

Cette méthode suppose que la fenêtre graphique est déjà ouverte.

Exercice 11 — *Visualisation* Ajoutez à la classe `Camera` une méthode de visualisation d'un ensemble de figure :

```
public void visualiser(Figure [] figures)
```

Cette méthode devra ouvrir la fenêtre graphique puis dessiner l'ensemble des figures passées en paramètre en les projetant sur l'écran de la `Camera` courante.

Indication — *Vous pouvez définir une méthode qui permette la visualisation d'un segment et une méthode qui permette la visualisation d'une figure. Dans ce cas vous auriez le choix entre les deux solutions suivantes :*

[2] voir l'exercice 20 pour une version plus complexe.

– *ajouter les méthodes suivantes à la classe* `Camera` :

```
public void visualiser(Segment s)
public void visualiser(Figure f)
```

– *ajouter la méthode suivante à la classe* `Segment` *et à la classe* `Figure` :

```
public void visualiser(Camera c)
```

Exercice 12 — *Programme principal* Complétez la classe `Exemple` de la question 3 en créant une `Camera` puis en lui faisant afficher la pyramide que vous avez créée.

Créez ensuite une seconde `Camera` pour visualiser la pyramide depuis un autre angle.

INDICATION — *Si vous avez utilisé les coordonnées proposées à la question 4 pour la pyramide, vous pouvez utiliser les paramètres suivants pour la première* `Camera` :

$$\mathtt{W} = (-1.2, -1, 0) \quad \mathtt{C} = (150, 120, 100) \quad \mathtt{d} = 250 \quad \mathtt{l} = 800 \quad \mathtt{h} = 450$$

Exercice 13 — *Objets situés derrière la `Camera`* Le système de tracé actuel affiche tous les objets sur l'écran, y compris ceux qui sont situés « derrière » la caméra, par rapport au plan $(C, \vec{u}, \vec{v})$. Ajoutez à la classe `Camera` une méthode qui teste si un `Point` est situé devant la caméra :

```
public boolean estDevant(Point p)
```

Modifiez la méthode d'affichage d'un segment pour qu'il n'affiche le `Segment` que si ses deux extrémités sont situées devant la `Camera`.

INDICATION — *Pour tester si un* `Point M` *est devant la caméra, on peut regarder le signe du produit scalaire* $\overrightarrow{CM}.\vec{w}$.

Que faire si seulement l'une des deux extrémités d'un segment est devant la caméra ?

(...)

La mécanique est maintenant en place pour pouvoir décrire une scène et puis l'afficher. Cependant décrire une scène `Segment` par `Segment` peut se révéler un travail fastidieux et répétitif ; ou bien encore : trouver les bonnes coordonnées pour la `Camera` peut demander de nombreux essais et presque autant erreurs. La suite du TP propose une idée pour simplifier la création de figures puis une autre idée pour écrire un programme interactif de déplacement d'une `Camera`. Les exercices suivants sont facultatifs, et vous pouvez traiter les deux problèmes dans l'ordre que vous voulez.

Création de figure

Transformation géométrique d'une figure

Vous pouvez maintenant écrire quelques méthodes de transformation géométrique sur les figures. En cas de doute sur les formules mathématiques n'hésitez pas à consulter Wikipedia[3].

RAPPEL — *Les transformations suivantes sont toutes des similitudes[4]. En particulier, il suffira pour transformer une figure d'appliquer la transformation à chacun des points qui la composent.*

Exercice 14 — *Translation* Ajoutez à la classe `Figure` une méthode qui permette de déplacer une figure dans l'espace. Elle pourrait avoir la signature suivante :

```
void translation(Vecteur t)
```

INDICATION — *Vous pouvez aussi ajouter une méthode* `translation` *à la classe* `Segment` *et à la classe* `Point`.

[3] http://fr.wikipedia.org/wiki/Transformation
[4] http://fr.wikipedia.org/wiki/Similitude

Exercice 15 — *Homothétie* Ajoutez à la classe `Figure` une méthode qui agrandisse la figure courante. Pour cela vous pourrez ajouter les méthodes suivantes :

```
void homothétie(double r) // homothétie de centre O.
void homothétie(Point centre, double rapport)
```

Exercice 16 — *Rotation (pour les motivés)* Ajoutez à la classe `Figure` une méthode qui permette tourner une figure. Pour cela vous pourrez ajouter les méthodes suivantes :

```
void rotation(Vecteur axe, double angle) // rotation de centre O.
void rotation(Point centre, Vecteur axe, double angle)
```

Exercice 17 — *Copie d'une figure* Comment feriez-vous pour copier une figure afin qu'appliquer une transformation géométrique à la figure originale ne modifie pas la copie ?
Serait-il utile de définir une méthode de copie d'un vecteur ?

Figures de base

Exercice 18 — *Cube* Créez une classe `FigureDeBase` avec une méthode statique permettant de construire un cube de centre (0,0,0) et de côté 20 :

```
public static Figure cube()
```

Ajoutez ensuite une seconde méthode statique permettant de construire un cube dont le centre et le côté sont passés en argument :

```
public static Figure cube(Point centre, double cote)
```

Indication — *Vous pouvez commencer par créer un carré de côté 20 et de centre (0,0,0) puis lui appliquer deux transformations géométriques.*

Exercice 19 — *Autres figures* Ajoutez, comme pour le cube, des méthodes statiques permettant de créer une pyramide à base carré ou triangulaire, un parallélépipède rectangle, un cylindre dont la base est un polygone régulier ...

Amélioration de la classe `Camera`

Faciliter la création d'une `Camera`

Exercice 20 — *Amélioration de la méthode* `update` Dans la question 7 on supposait que le vecteur $\vec{w}$ était parallèle au plan $(0, \vec{i}, \vec{j})$. Reprenez la méthode `update` sans faire cette hypothèse, mais en supposant que le vecteur $\vec{v}$ appartienne au plan $(X, \vec{k}, \vec{w})$. Si les vecteurs $\vec{k}$ et $\vec{w}$ sont colinéaires, vous pourrez prendre $\vec{v} = -\vec{i}$.

Indication — *N'hésitez pas à profiter des propriétés géométriques du produit vectoriel.*

Que doit-on faire si l'on veut pas non plus faire d'hypothèse sur $\vec{v}$?

Exercice 21 — *Second constructeur* Ajoutez à la classe `Camera` un second constructeur qui prenne le centre X de l'écran en argument à la place du vecteur W et de la distance d.

```
public Camera(Point c, Point X, int l, int h)
```

Déplacement interactif d'une `Camera`

Exercice 22 — *Déplacement* Ajoutez à la classe `Camera` une méthode permettant de déplacer la caméra courante en appliquant une translation à son point `C`. Cette méthode devra ensuite réafficher la scène.

```
public void deplacer(Vecteur v)
```

Exercice 23 — *Rotation* Ajoutez à la classe `Camera` une méthode permettant de faire tourner l'axe de vision autour de la droite $(C, \vec{v})$, puis de réafficher la scène :

```
public void rotationC(double angle)
```

INDICATION — *Si vous ajoutiez une méthode* `rotation` *à la classe* `Vecteur`*, il suffirait alors d'appliquer une certaine rotation au vecteur* `W` *et de mettre à jour* `X`*,* `U` *en conséquence .*

Exercice 24 — *Rotation (bis)* Ajoutez à la classe `Camera` une méthode permettant de faire tourner une caméra autour de la droite $(X, \vec{v})$ puis de réafficher la scène :

```
public void rotationX(double angle)
```

INDICATION — *Vous pouvez dans un premier temps calculer le nouveau vecteur* `W`*, puis en déduire le nouveau point* `C` *et le nouveau vecteur* `U`*.*

Exercice 25 — *Programme interactif* Modifiez la fonction de visualisation afin que la saisie de certain caractère au clavier permette par exemple :

- d'avancer la caméra de 20 unités,
- de faire tourner l'axe de vision de 0.1 radian autour de la droite $(C, \vec{v})$,
- de faire tourner la caméra de 0.1 radian autour de la droite $(X, \vec{v})$,

puis de réafficher la scène.