TP d'APOO, 1ère partie
Visualisation d'objets 3D

Cette première partie du TP porte sur la représentation et la visualisation d'objets 3D.

La première section explique le processus de génération de l'image 2D, affichable à l'écran, de la scène 3D observée. La seconde détaille le travail demandé. Enfin la troisième fournit les compléments mathématiques nécessaires.

1 Principes de la visualisation 3D

Cette partie ne comporte pas de question. Elle donne des informations utiles à la compréhension du problème.

1.1 Scène 3D

La scène à visualiser est un ensemble de modèles 3D. Chacun de ces modèles est représenté par un maillage surfacique, c'est-à-dire un ensemble de facettes **triangulaires** (voir la figure 1). Les coordonnées des sommets de ces facettes sont exprimées dans un unique référentiel Ref_{scene}.

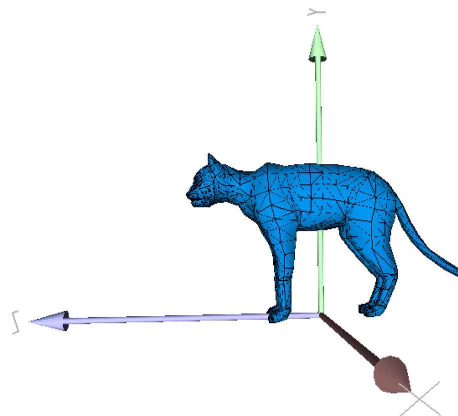


Figure 1: Un modèle de chat triangulé, dans un référentiel Ref_{scene}.

L'affichage de la scène complète consiste à afficher toutes les facettes de chacun des modèles. L'affichage d'une facette se ramène simplement à opérer un changement de référentiel 3D suivi d'une projection 3D/2D pour chaque sommet de la facette, opérations qui se réalisent par des multiplications de matrices. Les facettes doivent être affichées de la plus distante à la plus proche. L'affichage écran (clipping, discrétisation) ne sera pas à votre charge.

1.2 Caméra : de la 3D au plan image 2D

Pour générer l'image 2D à l'écran, la scène est observée par une caméra. La modélisation de cette caméra repose sur une opération dite de projection 3D/2D.

1.2.1 Projection 3D/2D

L'opération de *projection* permet de déterminer les coordonnées 2D de l'image d'un point de l'espace 3D. Les plus courantes sont :

la projection perspective ou projection centrale (figure 2) : chaque point est projeté en direction du point d'observation de la scène O, aussi appelé centre de projection, situé à une distance déterminée du plan image (la distance focale f).

La projection A' d'un point A est l'intersection du segment $[OA]$ et du plan image.

la projection parallèle (figure 3) : le centre de projection est supposé à une distance infinie du plan de projection. Chaque point se projette parallèlement à une unique direction de projection, généralement perpendiculaire au plan image (projection orthographique).

La projection d'un point est l'intersection de la parallèle à la direction de projection passant par ce point et du plan image.

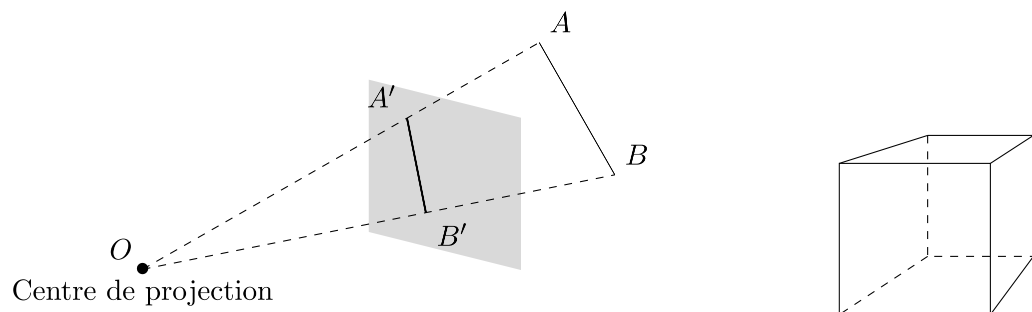


Figure 2: Projection perspective : tous les points se projettent vers le centre de projection. Les objets sont déformés, mais cette projection est la plus réaliste par rapport à la vision humaine.

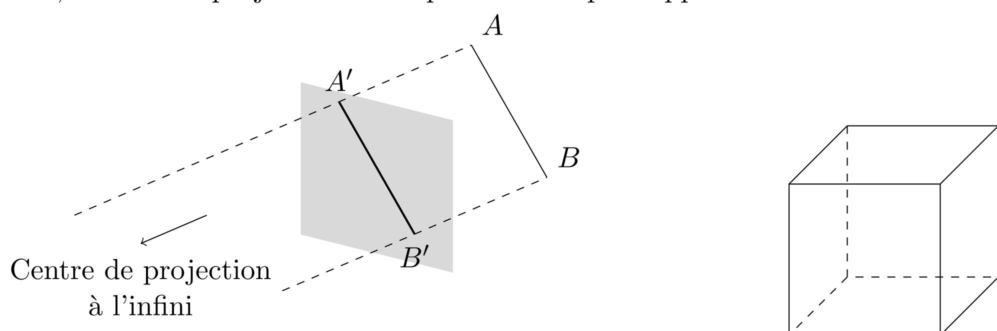


Figure 3: Projection parallèle : le centre de projection est considéré comme à l'infini. Les rapports de distances sont conservés, mais l'apparence est moins réaliste.

Ces projections conservent l'alignement et l'intersection. Mais contrairement à la projection parallèle, la projection centrale ne conserve ni le parallélisme ni les rapports de distance.

En pratique, la représentation en perspective "déforme" les objets mais assure que si deux objets de même taille sont observés, le plus éloigné apparaît plus petit à l'écran. C'est la

représentation la plus proche de la réalité perçue par l'oeil. La projection parallèle est elle essentiellement utilisée pour les applications nécessitant une mesure des distances.

1.2.2 Représentation d'une caméra et image d'un point

En définissant un référentiel Ref_{camera} lié à la caméra, l'opération de projection peut être représentée par une matrice homogène M_{proj} (voir section 3.5). Dans ce référentiel, la projection d'un point P_{camera} est simplement le produit $M_{\text{proj}} \cdot P_{\text{camera}}$.

Pour utiliser ce formalisme, il est donc nécessaire d'exprimer les coordonnées des modèles, initialement dans Ref_{scene}, dans le référentiel de la caméra. Ce changement de référentiel est en fait lié à la *position* de la caméra par rapport à la scène. Là encore, cette position peut être représentée par une unique transformation homogène, $M_{\text{sceneAcamera}}$, composition de rotations et translations (section 3). Dans le cadre de ce TP, cette position peut être modifiée via les interactions de l'utilisateur, notamment avec la souris gérée par l'IHM qui vous est fournie.

Une caméra est donc caractérisée par une matrice de projection et une matrice donnant sa position par rapport à la scène. Si P_{scene} est un point 3D dans le référentiel de la scène, son image dans le plan 2D de la caméra est alors:

$$P_{\text{image}} = M_{\text{proj}} \cdot M_{\text{sceneAcamera}} \cdot P_{\text{scene}}$$

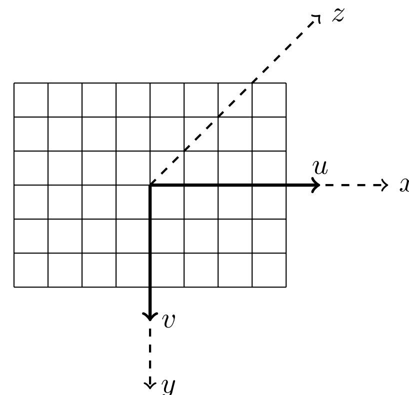
Seules les coordonnées x et y sont pertinentes (z est seulement la position du plan image de la caméra dans P_{camera}), donc P_{image} peut être considéré comme un point 2D.

1.3 Affichage de la scène

1.3.1 Référentiel discret de l'écran

Un point P_{image} est à coordonnées réelles, dans un plan 2D supposé infini. Or une image à l'écran est *discrète*. Elle peut être vue comme une grille 2D de pixels, "cases" 2D de taille finie et de couleur unique.

Un dernier référentiel Ref_{ecran}, 2D et discret, est donc introduit. L'axe des abcisses discrètes u est horizontal, et l'axe des ordonnées discrètes v est vertical, dirigé vers le bas.
Cette configuration, qu'on retrouve dans tous les systèmes informatiques, revient en fait à considérer que l'axe y de la caméra pointe vers le bas, et donc que l'axe z traduit la distance à l'écran (plus z est grand, plus un point est éloigné). On suppose également que l'axe z intercepte le plan image au milieu de l'écran.



Pour le TP on choisit de fixer la taille de l'image, soit le nombres de pixels horizontaux et verticaux. Pour associer un point P_{image} au pixel correspondant, il est aussi nécessaire de fixer la taille du pixel par rapport à l'unité de mesure du référentiel dans lequel les objets sont définis.

1.3.2 Image d'une facette 3D

Une facette triangulaire est définie dans le référentiel 3D de la scène, par trois sommets $S1_{\text{scene}}$, $S2_{\text{scene}}$ et $S3_{\text{scene}}$. Son image par projection est un triangle dont les sommets 2D sont les projetés $S1_{\text{image}}$, $S2_{\text{image}}$ et $S3_{\text{image}}$. A ces points sont associés trois pixels $S1_{\text{ecran}}$, $S2_{\text{ecran}}$ et $S3_{\text{ecran}}$.

Le tracé à l'écran de ce triangle nécessite deux étapes:

- *Rasterisation*: consiste à déterminer, à partir des trois pixels sommets, la couleur de chaque pixel de la grille pour dessiner au mieux l'image du triangle.
- *Clipping*: le champ de vision de la caméra est en fait une aire rectangulaire finie, de dimension `DimU` par `DimV`. Seule la partie visible du triangle (qui est de manière générale un polygone convexe) doit bien sûr être affichée

Ces deux étapes ne seront pas traitées dans ce TP[1]. Elles seront ici réalisées à l'aide des opérations graphiques du langage Java, en particulier les méthodes `drawPolygon` et `fillPolygon` de la classe `Graphics`.

1.3.3 Image de la scène complète : ordre d'affichage des facettes

Toutes les facettes de la scène ne sont pas visibles : certaines peuvent être opposées à la caméra ou masquées par d'autres objets. Pour un résultat correct, l'ordre d'affichage des images des facettes est donc important car une image peut être, au moins partiellement, recouverte par les images affichées après elle. L'algorithme dit "du peintre" consiste à afficher en premier l'image des facettes les plus éloignées de la caméra. Ainsi les facettes des parties visibles des objets, les plus proches de la caméra, ne seront pas recouvertes.

Pour réaliser ceci, avec une position de caméra donnée :

- une *altitude* suivant l'axe z est associée à chaque image de facette. Il s'agit par exemple de la coordonnée z du barycentre de la facette 3D dans Ref_{camera}.
- les images de facettes sont ensuite affichées par ordre décroissant selon cette altitude.

[1] mais elles devraient rappeler quelque chose à ceux qui ont suivi le cours optionnel *Interfaces Utilisateur Graphiques* l'an dernier ...

2 Travail à réaliser

Vous aurez à écrire des classes réparties dans trois packages : `geometrie`, `scene` et `vision`. Une base de code à compléter est également fournie, pour gérer l'interface graphique du TP.

2.1 Package `geometrie`

Ce premier package doit fournir :

- une classe `Point` représentant les points de l'espace 3D.
- une classe `Transformation` permettant de représenter les différentes transformations (rotations, projections 3D/2D, ...) décrites en section 3, et les opérations nécessaires.

  L'initialisation des transformations, par exemple une rotation d'angle `angle` autour de l'axe des x, pourra se faire à l'aide d'une méthode d'instance de type :

  ```
  public void setToRotationX(double angle);
  ```

  ou d'une méthode de classe de type :

  ```
  public static Transformation rotationX(double angle);
  ```

- un programme de tests unitaires de ces classes, pour les valider.

2.2 Package `scene`

Ce package peut n'être écrit que partiellement au début (avec un seul type simple de modèle, par exemple `Cube`) pour valider la chaîne de visualisation.

Le package `scene` regroupe les classes permettant de définir une scène et les modèles qui la composent. Il doit fournir :

- une classe `Facette3D` définissant une facette ;
- une classe `Scene`, composée d'un ensemble de modèles 3D ;
- une interface `Modele3D` déclarant une seule méthode :

```
public interface Modele3D {
    /**
     * Retourne l'ensemble des facettes composant le modele,
     * dans un ordre indefini.
     */
    public Iterator<Facette3D> getFacettes();
}
```

  Cette interface permet une manipulation *uniforme* de tous les types de modèles depuis le package `vision`. Si cela s'avère nécessaire (en fonction de votre implémentation), vous pouvez définir une classe abstraite au lieu d'une interface. Justifiez.

- quelques classes simples implémentant l'interface `Modele3D` permettant de construire des objets simples : `Cube`, `Cylindre`, ... ;
- des classes spécifiques d'objets construits à partir d'instances de formes simples auxquelles une transformation aura été appliquée.

  Par exemple, on peut construire une classe `HaltereCarree` à partir d'un `Cylindre` et un `Cube` à chaque extrémité ;

- une classe `ModeleMaillage`, qui construit un modèle à partir d'un fichier ASCII de description de maillage au format `.off`. Sous sa forme la plus simple, le format est le suivant :

```
OFF                         # en-tête OFF
NVert NFace  0              # nb de sommets, nb de faces
x y z                       # coordonnées du sommet 0
...
x y z                       # coordonnées du sommet NVert-1
Nv v[0] v[1] ... v[Nv-1]    # face 0 : Nv sommets dans la face
...                         #  (v[i] est un index de sommet dans 0..NVert-1)
Nv v[0] v[1] ... v[Nv-1]    # face NFace-1
```

  c'est à dire un entête `OFF`, suivi du nombre de primitives, de la liste des coordonnées des sommets et de la liste des faces. Chacune des faces comporte le nombre de sommets et une liste d'indices de sommets définissant la face. Dans ce TP, se limiter aux maillages à facettes triangulaires.

  Quelques maillages dans ce format vous seront fournis, qui peuvent être visualisés avec l'outil `geomview`.

2.3 Package `vision`

Ce package regroupe les classes fondamentales du processus de visualisation. Il devra contenir :

- une ou plusieurs classes représentant les différents modèles de caméra, perspective ou parallèle, réalisant les opérations de projection 3D/2D d'un point.

- une classe `ImageFacette` qui gère les données permettant d'afficher l'image d'une facette : coordonnées discrètes des sommets, altitude.

- une classe `AfficheurScene` qui contient la scène et une caméra et génère les données à fournir à l'IHM pour l'affichage.

  Pour pouvoir être intégrée par l'IHM, cette classe doit implémenter l'interface `Affichable` :

```java
/**
 * Une classe implementant cette interface peut etre utilisee
 * dans l'application du TP: visualisation de facettes, avec
 * vue controlee par une camera.
 */
public interface Affichable {
    /**
     * Retourne l'ensemble des images de facettes 3D a afficher.
     * L'ordre d'affichage est celui du parcours de l'iterateur.
     */
    public Iterator<ImageFacette> getImagesFacettes();

    /**
     * Definit la position de la camera.
     * @param t nouvelle position, remplace la position actuelle.
     */
    public void positionneCamera(Transformation t);

    /**
     * Met la camera en mode de projection orthographique.
```

```
     */
    public void setToCameraOrthographique();

    /**
     * Met la camera en mode de projection perspective.
     * @param angle l'angle d'ouverture de la camera.
     */
    public void setToCameraPerspective(int angle);
}
```

Dans un premier temps, les méthode de gestion de la caméra peuvent être "désactivées" en les définissant dans `AfficheurScene` comme ne faisant rien. Elles seront ensuite complétées avec l'avancement du TP.

2.4 Code fourni : package `ihm` et programme principal

Le package `ihm` regroupe les classes de l'interface graphique du TP : une fenêtre de dessin, la gestion d'évènements souris pour déplacer la caméra autour de la scène, et quelques boutons permettant de choisir le type de caméra utilisé ou de remettre la caméra dans sa position initiale.

Seule la méthode `paintComponent(Graphics g)` de la classe `ZoneDessin` doit être complétée. C'est ici que vous devez effectivement dessiner les images des facettes, à l'aide des méthodes de `Graphics` (`fillPolygon`, ...). Cette opération est directement dépendante de votre classe `ImageFacette`.

Les autres classes du package n'ont pas besoin d'être modifiées.

Une dernière classe `TPVisu3D` est fournie, contenant le programme principal. Sa méthode `creeScene()` devra être écrite pour construire la scène à afficher dans votre TP.

3 Annexe : transformations 3D

Cette annexe fournit les compléments mathématiques sur les transformations. Vous pouvez passer rapidement si vous le voulez, et implémenter les formules "en aveugle" : la compréhension de cette partie ne fait pas partie du programme d'APOO.

De nombreuses transformations interviennent dans le processus de visualisation : positionnement d'objets ou de caméras, changements de référentiels, ... Le cas particulier des transformations de projection 3D/2D est traité section 3.5.

3.1 Transformations en coordonnées cartésiennes

Une représentation usuelle des transformations dans l'espace cartésien est sous forme matricielle :

Translation $P' = T + P$

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} + \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Homothétie $P' = S \cdot P$

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Rotation autour de O_x $P' = R_x \cdot P$

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

La composition de transformations est difficile sous cette forme, les opérations n'étant pas toujours de même nature (somme pour la translation, produit pour les rotations). Pour cette raison, la notation en *coordonnées homogènes* est préférée.

3.2 Transformations en coordonnées homogènes

3.2.1 Coordonnées homogènes

Un point d'un espace de dimension n est représenté en coordonnées homogènes par $n + 1$ coefficients. En 3D, le point cartésien (x, y, z) est ainsi représenté par un vecteur de dimension 4 (sx, sy, sz, s).

Les coordonnées d'un point sont en fait définies à un facteur multiplicatif près, supposé non nul. Ainsi $(2, 4, -1, 1)$ et $(6, 12, -3, 3)$ sont deux représentations homogènes du même point. Pour retrouver l'unique point cartésien correspondant, il suffit de diviser chaque coordonnée par le facteur s, c'est-à-dire la quatrième coordonnée : $(x/s, y/s, z/s)$.

3.2.2 Matrices homogènes de transformation

En coordonnées homogènes, *toutes* les transformations peuvent être représentées par une matrice M de dimension $n + 1$. La transformation d'un point P est alors simplement obtenue par le produit $P' = M \cdot P$.

Les matrices homogènes de transformation usuelles sont définies figure 4.

Translation : $T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Homothétie : $S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Rotations autour des axes principaux, avec $c = \cos\theta$ et $s = \sin\theta$:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_y(\theta) = \begin{bmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_z(\theta) = \begin{bmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation autour d'un axe $n(x, y, z)$, avec $c = \cos\theta$, $s = \sin\theta$ et $k = 1 - c$:

$$R(n, \theta) = \begin{bmatrix} kx^2 + c & kyx - sz & kzx + sy & 0 \\ kxy + sz & ky^2 + c & kzy - sx & 0 \\ kxz - sy & kyz + sx & kz^2 + c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 4: Matrices homogènes des transformations affines standard.

3.3 Composition de transformations

En coordonnées homogènes, la composition de transformations est très simple: il suffit de multiplier les matrices. Par exemple, le résultat de la rotation d'un point autour de l'axe O_x suivie d'une translation T est obtenu par $P' = T \cdot (R_x \cdot P) = (T \cdot R_x) \cdot P$.

De manière générale, une transformation composée possède toujours une partie rotation et un partie translation, de la forme:

$$\begin{bmatrix} \begin{pmatrix} & & \\ & R & \\ & & \end{pmatrix} & \begin{pmatrix} \\ T \\ \\ \end{pmatrix} \\ 0 \quad 0 \quad 0 & 1 \end{bmatrix}$$

Attention :

- les matrices de transformation apparaissent dans le produit dans l'ordre inverse de l'ordre selon lequel elles interviennent effectivement. Sur l'exemple $(T \cdot R_x)$, la rotation a lieu en premier puis la translation ;
- la composition n'est pas commutative ! Faites un schéma, $(T \cdot R_x) \cdot P$ et $(R_x \cdot T) \cdot P$ ne donnent pas le même résultat.

3.4 Inversion

Pour les transformations affines ci-dessus, l'inversion d'une transformation consiste à :

1. inverser la partie rotation, soit la matrice (3x3) en haut à gauche de la matrice homogène. L'inverse d'une matrice de rotation R est simplement sa transposée : $R^{-1} = R^T$;
2. inverser la partie translation, soit le vecteur T (3x1) en haut à droite de la matrice homogène. Son inverse est le produit de la rotation inverse vecteur par $-T$: $-R^{-1}T$

$$\text{L'inverse de } \begin{bmatrix} \begin{pmatrix} & R & \end{pmatrix} & \begin{pmatrix} T \end{pmatrix} \\ 0 \quad 0 \quad 0 & 1 \end{bmatrix} \text{ est donc } \begin{bmatrix} \begin{pmatrix} & R^{-1} & \end{pmatrix} & \begin{pmatrix} -R^{-1}T \end{pmatrix} \\ 0 \quad 0 \quad 0 & 1 \end{bmatrix}$$

3.5 Représentation des projections

Pour travailler sur les projections, on introduit un référentiel Ref_{camera} dont l'axe z est orthogonal au plan sur lequel se forme l'image. Soit $P(x, y, z)$ un point dans les coordonnées sont exprimées dans ce référentiel. Plus la valeur de z est grande, plus P est éloigné du point d'observation. Le calcul de l'image P' de P est bien sûr dépendant du type de projection.

Le formalisme des matrices homogènes permet également de représenter les transformations perspective ou parallèle. Ainsi, il est possible de combiner des transformations affines puis une projection par simple composition de matrices.

Les matrices de projection ne sont par contre pas inversibles. La source d'un point image se trouve sur une droite, mais on ne peut pas dire où...

3.5.1 Matrice de projection perspective

Le centre de projection est généralement à l'origine du référentiel, et le plan image à l'altitude $z = f$, où f est la distance focale (figure 5). A partir des relations $\frac{z}{f} = \frac{x}{x'}$ et $\frac{z}{f} = \frac{y}{y'}$, les coordonnées de la projection de P sont : $P' = (x', y', z') = \left(\frac{x}{z/f}, \frac{y}{z/f}, f\right)$

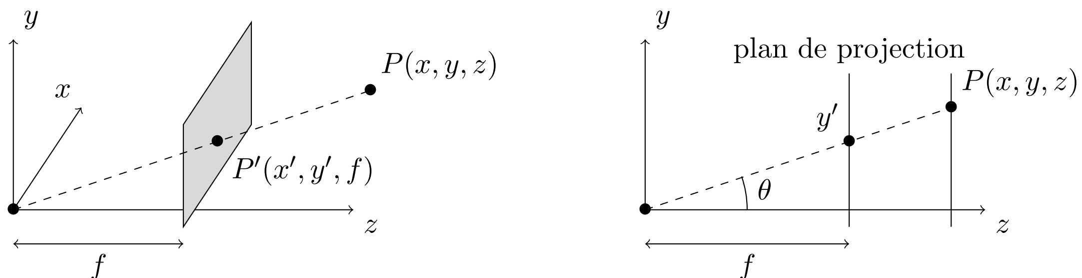


Figure 5: Calcul des coordonnées en projection perspective. A droite, vue dans le plan yz. L'angle θ est appelé l'*ouverture* de la caméra.

La projection en perspective peut être représentée par la matrice en coordonnées homogènes :

$$M_{\text{persp}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/f & 0 \end{bmatrix}$$

L'image P' d'un point P est simplement obtenu par le produit :

$$P' = M_{\text{persp}} \cdot P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/f & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z/f \end{bmatrix}$$

On remarquera que le facteur multiplicatif du point homogène $P' = (x, y, z, z/f)$ n'est plus égal à 1, mais le point cartésien associé est bien le point de projection recherché.

3.5.2 Matrice de projection parallèle

La direction de projection est l'axe z, et par convention le plan image est à l'altitude $z = 0$ (figure 6). Le choix de cette altitude n'influence que la partie visible de la scène (en avant du plan image), pas les coordonnées projetées.

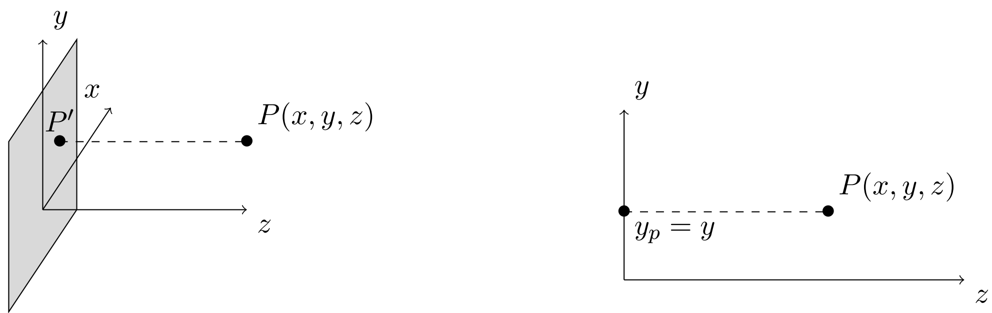


Figure 6: Calcul des coordonnées en projection parallèle. Il suffit d'annuler la composante z.

Le projeté P' de P s'obtient simplement en annulant la coordonnée z : $P' = (x', y', z') = (x, y, 0)$. La projection parallèle est représentée en coordonnées homogènes par la matrice :

$$M_{\text{paral}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

TP d'APOO, 2nde partie
Rebouchage de trous dans des objets 3D

Cette seconde partie du TP porte sur le rebouchage de trous dans des objets 3D maillés, grâce à une méthode de programmation dynamique.

La première section explique le problème posé, et détaille sa modélisation mathématique. La seconde détaille le travail demandé. Enfin la troisième fournit une introduction à la programmation dynamique. *Attention : la programmation dynamique fait partie du programme d'APOO (vous êtes susceptibles d'en retrouver à l'examen écrit) ; la compréhension de la troisième section est donc requise.*

1 Modélisation du problème

Cette partie ne comporte pas de question.

En infographie, les surfaces constituées de facettes sont souvent modélisées à partir d'objets réels. Cependant, les surfaces ainsi créées peuvent comporter des trous, qu'il convient de boucher avec un ensemble de facettes triangulaires (voir la figure 1).

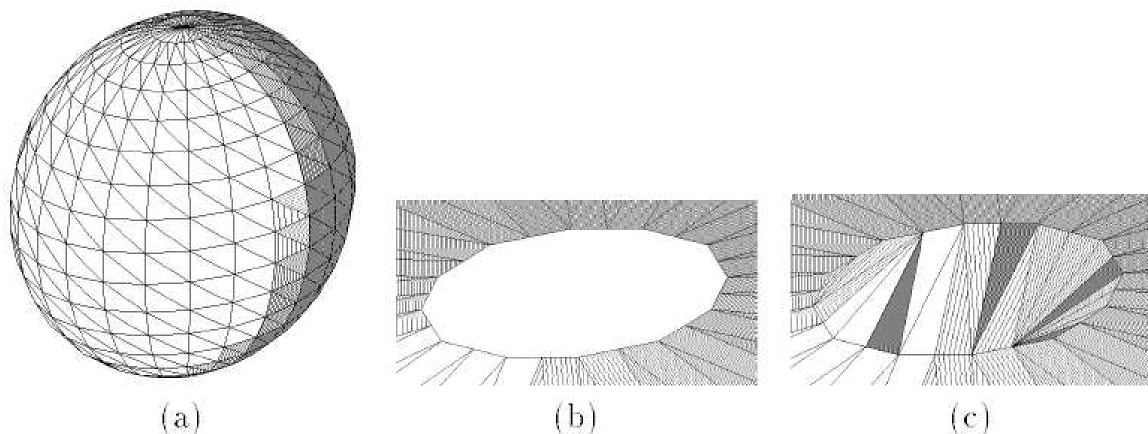


Figure 1: (a,b) Sphère trouée, et (c) rebouchage du trou.

Le trou à reboucher est défini par une suite circulaire $P_0, P_1, \ldots, P_{n-1}, P_n = P_0$ de `Point`, formant un polygone non nécessairement plan. On va chercher une *triangulation* de ce polygone, c'est-à-dire une collection de facettes triangulaires dont les sommets coïncident avec ceux du polygone. Plus exactement, on va chercher une triangulation qui *minimise* l'aire totale $\mathcal{A}$ des facettes, afin d'obtenir un rebouchage le plus plat possible. On notera $\mathcal{A}_{i,j}, 0 \leq i < j \leq n-1$, l'aire totale pour le sous-polygone constitué uniquement à partir des points $P_i, \ldots, P_j$. On a

donc $\mathcal{A} = \mathcal{A}_{0,n-1}$.

Rappel : l'aire d'une facette définie par les trois `Point` P_1, P_2 et P_3 est donnée par (formule de Héron) :

$$Aire(P_1, P_2, P_3) = \sqrt{p(p-a)(p-b)(p-c)}$$

avec a, b et c les longueurs des trois côtés de la facette ($a = \|P_1P_2\|, b = \|P_2P_3\|, c = \|P_3P_1\|$) et p le demi-périmètre ($p = \frac{a+b+c}{2}$).

2 Travail à réaliser

On vous demande de proposer et comparer trois solutions à ce problème, qui vous sont détaillées ci-après.

2.1 Solution récursive naïve

Les réponses aux questions 1, 2, 3 et 5 ci-dessous sont à fournir dans le rapport.

1. Montrer que, $\forall i \in [0, n-2], \mathcal{A}_{i,i+1} = 0$, et que, $\forall i \in [0, n-3], \mathcal{A}_{i,i+2} = Aire(P_i, P_{i+1}, P_{i+2})$.
2. Soit $0 \leq i < j \leq n-1$. Montrer que :
$$\mathcal{A}_{i,j} = \min_{i<k<j} (\mathcal{A}_{i,k} + \mathcal{A}_{k,j} + Aire(P_i, P_k, P_j)). \quad (1)$$
3. Dessiner le graphe de dépendances de $\mathcal{A} = \mathcal{A}_{0,n-1}$. Y a-t-il des calculs redondants ?
4. Créer une nouvelle classe `Rebouchage`. Ajouter à cette classe une méthode statique *récursive*
```
static double rebouchageTrouRecNaif(Point[] polygone);
```
qui donne **l'aire** de la triangulation d'aire minimale d'un polygone donné en paramètre sous forme d'un tableau de `Point`, grâce à la formule (1) ci-dessus.
5. Quelle est la complexité en pire cas de cette méthode ?

2.2 Solution récursive avec mémoire cache

La réponse à la question 2 ci-dessous est à fournir dans le rapport.

1. Ajouter à la classe `Rebouchage` une méthode *récursive avec mémoire cache*
```
static double rebouchageTrouRecCache(Point[] polygone);
```
qui utilise une mémoire cache (voir annexe) pour résoudre le problème sans calculs redondants.
2. Quelle est la complexité en pire cas de cette méthode ? Quelle est la place mémoire nécessaire pour la mémoire cache ?

2.3 Solution itérative

Les réponses aux questions 1 et 3 ci-dessous sont à fournir dans le rapport.

1. Donner un tri topologique du graphe de dépendances permettant d'obtenir une solution *itérative* au problème. Décrire cette solution.
2. Ajouter à la classe `Rebouchage` une méthode *itérative*
```
static double rebouchageTrouIter(Point[] polygone);
```
qui résoud le problème sans calculs redondants.
3. Quelle est la complexité en pire cas de cette méthode ? Quelle est la place mémoire nécessaire ?

2.4 Tests, comparaison et analyse des trois solutions

La réponse à la question 2 ci-dessous est à fournir dans le rapport.

1. Pour tester vos méthodes, créer par exemple une classe `Cone` implémentant l'interface `Modele3D`, permettant de représenter un cône constitué de facettes, sans sa base. Pour cela, partir de n `Point` $p_1, p_2, \ldots, p_n$ formant un polygone (*pas forcément plan*), et relier tous ces points à un même autre point p_0 (le sommet du cône) : le cône est l'union des facettes (p_0, p_i, p_{i+1}). La surface à trianguler est la base du cône, c'est-à-dire le polygone $p_1, \ldots, p_n$.

2. Tester le temps de calcul des trois méthodes `rebouchageTrouRecNaif`, `rebouchageTrouRecCache` et `rebouchageTrouIter` sur des trous de tailles (= nombre de points) différentes. Le résultat est-il conforme aux complexités en pire cas que vous avez calculées ? Qu'en pensez-vous ?

2.5 Calcul de la triangulation

1. Modifier la méthode `rebouchageTrouRecCache` afin de calculer effectivement la meilleure triangulation du polygone.

2. Ajouter une méthode dans l'interface `Modele3D` qui prend en paramètre un polygone défini par une liste de `Point`, et qui ajoute au modèle la triangulation de ce polygone.

3 Cours : introduction à la programmation dynamique

3.1 Programmation dynamique : principe général

3.1.1 Arbre d'appels de fonctions

L'arbre d'appels de fonctions est une représentation des étapes d'un calcul : chaque nœud représente un appel de fonction, les arcs indiquent l'imbrication des appels. Nous étiquetons un nœud avec les paramètres effectifs de l'appel, et un arc, éventuellement, avec la valeur du résultat.

Propriétés :

- lors de l'exécution de l'appel-racine, l'arbre est "développé" arc par arc : les états successifs de la pile d'exécution énumèrent tous les chemins partant de la racine ; autrement dit, l'état de la pile correspond à un chemin dans l'arbre, partant de la racine ;
- la taille de l'arbre est une mesure du temps d'exécution de l'appel-racine.

Voir la figure 2 pour un exemple d'arbre d'appels d'une fonction.

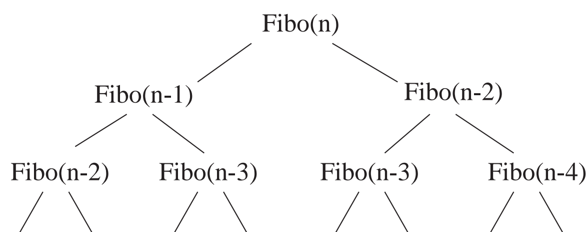


Figure 2: Arbre des appels d'une procédure récursive calculant la suite de Fibonacci.

3.1.2 Problème des calculs redondants

Dans cette annexe, nous allons étudier des décompositions récursives pour lesquelles une programmation directe, trop naïve, conduit à des inefficacités grossières. Nous introduisons alors plusieurs transformations de programmes, à caractère général, permettant d'aboutir à des algorithmes efficaces : c'est ce qu'on appelle la *programmation dynamique.*

Exemple 1 *Suite de Fibonacci : si on programme directement la relation de récurrence*

$$Fibo(n) = Fibo(n-1) + Fibo(n-2),$$

on effectue un grand nombre de calculs redondants (de `Fibo(n-3)`*,* `Fibo(n-4)`*, etc.), comme le montre l'arbre des appels de la figure 2. Et on obtient au bout du compte une complexité exponentielle !*

Exemple 2 *Le même phénomène se produit pour le calcul récursif des combinaisons C_n^p, définies par :*

$$C_n^p = C_{n-1}^p + C_{n-1}^{p-1}$$

Exercice d'échauffement : dessiner l'arbre des appels pour le calcul des C_n^p, et repérer les calculs redondants.

Remarque 1 *L'approche "diviser pour régner", que nous avons étudiée en algo 2 l'an dernier, échappe à ces problèmes de redondance, dans la mesure où il s'agissait de lancer le processus de résolution sur* deux moitiés disjointes *des données. Notons que c'est une approche "descendante" (top-down : descendante dans l'arbre d'appels de la fonction), par opposition aux méthodes ascendantes que nous allons introduire ci-dessous. Diviser pour régner est ainsi une méthode naturellement* récursive*, alors que l'élimination des calculs redondants conduira généralement à l'écriture d'algorithmes* itératifs*, voir section 3.1.5.*

3.1.3 Elimination des calculs redondants : mémoire cache

Un arbre d'appels de fonction peut donc mettre en évidence des calculs redondants : il y a un ou des calcul(s) redondant(s) si au moins un même sous-arbre apparaît plusieurs fois. Dans le cas de Fibonacci et des C_n^p, comme pour les problèmes traités dans la suite de cette annexe, les redondances sont énormes, aboutissant à des arbres de taille exponentielle (en la valeur d'un paramètre).

L'utilisation d'une "mémoire cache" (on parle aussi de *marquage*) est une méthode simple et efficace d'éliminer les calculs redondants : il s'agit de **mémoriser la valeur d'une fonction pour un jeu de paramètres donnés**, et de **consulter cette mémoire au début de la fonction**.

De façon générique, nous utilisons le cache avec 3 méthodes associées :

- `isComputed : params → bool`
- `getCache : params → val`
- `setCache : params, val → void`

Le cache est initialisé avec des valeurs "bidons", qui ne doivent pas faire partie des valeurs de retour possibles de la méthode (ou bien une valeur booléenne peut être utilisée en plus pour stocker l'état calculé ou non d'une valeur du cache). `isComputed()` renvoie vrai si et seulement si la valeur du cache pour un jeu de paramètres donné a été modifiée depuis l'initialisation (c'est-à-dire si la valeur stockée n'est plus la valeur "bidon"). `getCache()` permet d'accéder à la valeur du cache pour un jeu de paramètres donné. `setCache()` modifie le cache pour un jeu de paramètres donné, en remplaçant la valeur stockée par une nouvelle valeur entrée en paramètre de la méthode. **Généralement, `setCache()` modifie aussi la valeur renvoyée par `isComputed()` pour le même jeu de données.**

Remarque 2 *Cette transformation de programme est très simple, car on ne touche pas du tout au corps de la fonction initiale !*

La difficulté, bien entendu, est la gestion du cache. Dans les problèmes que nous traitons, un cache sera implanté par un tableau, à 1 ou 2 dimension(s).

Exemple 1: suite de Fibonacci

$$Fibo(n) = Fibo(n-1) + Fibo(n-2), \text{ avec } Fibo(1) = Fibo(2) = 1$$

Programmation directe :

```java
public int fibo(int n) {
    return (n <= 2) ? 1 : fibo(n-1) + fibo(n-2) ;
}
```

Avec cache :

```java
public int fibo(int n) {
    if (!isComputed(n))
        setCache(n, n <= 2 ? 1 : fibo(n-1) + fibo(n-2));
    return getCache(n);
}
```

Avec cette méthode, on ne calcule **qu'une fois** chaque `Fibo(i)` : la première fois. En effet, le premier appel à `Fibo(i)` met `isComputed(i)` à vrai et modifie la valeur de `cache(i)` : `cache(i)` ne contient plus une valeur "bidon", mais la valeur correcte (i.e. `Fibo(i-1)+Fibo(i-2)` si $i \geq 3$, 1 sinon). Lors des appels suivants à `Fibo(i)`, `isComputed(i)` vaut vrai et donc on renvoie directement la valeur de `cache(i)`.

Exemple 2 : combinaisons C_n^p

$$\begin{aligned} C_n^p &= C_{n-1}^p + C_{n-1}^{p-1} \quad \text{pour } 0 < p < n \\ C_n^0 &= C_n^n = 1 \end{aligned}$$

Ici on a besoin d'un cache géré par deux paramètres : n et p.
Programmation directe :

```java
public int c(int n, int p) {
    return (p == 0 || p == n) ? 1 : c(n-1,p) + c(n-1,p-1);
}
```

Avec cache :

```java
public int c(int n, int p) {
    if(!isComputed(n,p))
        setCache(n,p,(p == 0 || p == n) ? 1 : c(n-1,p) + c(n-1,p-1));
    return getCache(n,p);
}
```

Question 1 *Quel est l'effet de l'utilisation du cache sur l'arbre d'appels de fonctions ?*

Corrigé 1 *Tout nœud interne de l'arbre est unique. Si un nœud figure plusieurs fois dans l'arbre, l'occurrence la plus à gauche, la première dans l'ordre postfixée, est conservée. Les autres nœuds correspondent à des calculs redondants ; ils deviennent des feuilles, car le résultat est lu dans le cache.*

3.1.4 Graphe de dépendances des calculs et tri topologique

Dans un graphe de dépendances de calculs, chaque nœud représente un pas de calcul (par exemple un appel de fonction, ou une opération arithmétique), les arcs définissent les dépendances : le calcul de `Fibo(10)` a besoin des valeurs `Fibo(9)` et `Fibo(8)`. On obtient un graphe de dépendances à partir d'un arbre d'appels en **identifiant les nœuds ayant la même étiquette** ; le graphe de dépendances est homomorphe à l'arbre d'appels.

La figure 3 montre par exemple le graphe de dépendances pour Fibonacci : comparer avec l'arbre d'appels de la figure 2.

On remarque que la relation de dépendance est une *relation d'ordre partiel* sur les pas de calcul : pour pouvoir calculer `Fibo(10)`, il faut avoir calculé `Fibo(9)`, et donc `Fibo(9)` est *avant* `Fibo(10)` dans l'ordre des calculs. Le calcul récursif garantit que les cases du cache sont remplies

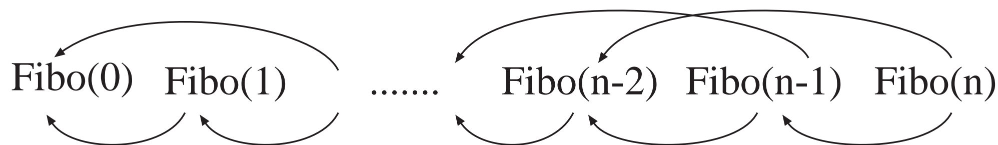


Figure 3: Graphe de dépendances pour Fibonacci.

dans un ordre total compatible avec cet ordre partiel ; il donne en fait un *tri topologique* des nœuds du graphe de dépendances (revoir le cours d'algo 2 de première année pour la définition d'un tri topologique).

3.1.5 Algorithmes itératifs

En fait, il y a autant de façons de calculer une fonction que de tris topologiques possibles de son graphe de dépendances. Une analyse approfondie du graphe peut mettre en évidence un ordre très simple – par exemple par un balayage du cache, de `cache(0)` à `cache(n)`. Ainsi on peut définir un algorithme *itératif* résolvant le problème initial : il suffit de faire les calculs dans l'ordre topologique du graphe des dépendances, en partant d'une feuille. Un algorithme itératif a souvent (mais pas tout le temps) l'avantage, par rapport à la résolution récursive avec cache, de diminuer la place mémoire nécessaire aux calculs. Le plus souvent, le temps de calcul reste inchangé : on fait les mêmes calculs, simplement pas dans le même ordre. Parfois cependant, le temps de calcul est allongé.

3.1.6 Programmation dynamique : méthodologie générale

L'objectif de la programmation dynamique est de résoudre un problème **récursif**, comportant des calculs redondants, de manière efficace en temps et en place mémoire. La méthodologie à utiliser en programmation dynamique est la suivante :

1. on détecte les calculs redondants en dessinant le graphe de dépendances de la fonction récursive : les arcs arrivant sur un même nœud correspondent à des calculs redondants ;
2. pour les éliminer, on a le choix entre deux types d'algorithmes :
   - calcul récursif avec marquage (i.e. mémoire cache) et stockage des résultats partiels, de manière à éviter de faire plusieurs fois le même calcul ;
   - calcul selon un ordre topologique sur les nœuds du graphe, ce qui a souvent l'avantage d'économiser la place mémoire. Ce procédé conduit à des algorithmes itératifs, qui, dans certains cas, peuvent être plus coûteux en temps.

   Alors que la première méthode peut être utilisée de manière systématique, la seconde correspond à une analyse fine du problème à résoudre de la part du programmeur.

3.2 Quelques exemples simples

3.2.1 Fibonacci

L'utilisation d'une mémoire cache a été détaillée plus haut ; on constate que le coût en temps de calcul devient linéaire au lieu d'être exponentiel ! La place mémoire nécessaire est en $O(n)$:

on utilise un tableau pour stocker toutes les valeurs des `Fibo(i)`.

L'écriture de l'algorithme itératif est triviale : il suffit de calculer les `Fibo(i)` par i croissant. On peut alors remarquer qu'il n'est en fait pas nécessaire de stocker les n valeurs calculées, mais seulement les deux dernières. Le coût mémoire est donc constant, égal à 2.

La version itérative est ici de coût identique à la version récursive avec cache, mais permet un gain de place mémoire.

3.2.2 Combinaisons C_n^p

Les nombres C_n^p sont donc définis par la formule de récurrence suivante :

$$\begin{aligned} C_n^p &= C_{n-1}^p + C_{n-1}^{p-1} \quad \text{pour } 0 < p < n \\ C_n^0 &= C_n^n = 1 \end{aligned}$$

Le graphe de dépendances est détaillé sur la figure 4 ci-dessous.

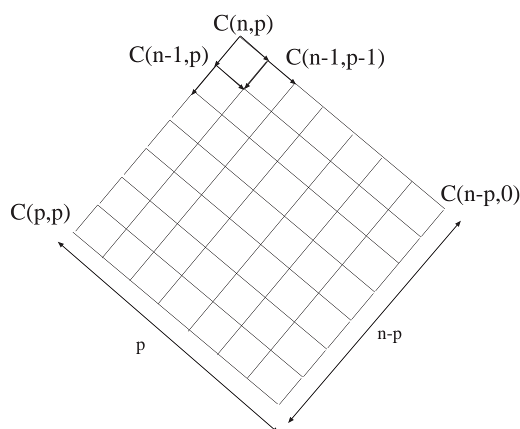


Figure 4: Graphe de dépendances d'un calcul récursif des C_n^p.

Ici, le cache peut être implanté par une matrice `M(p..n,0..p)` d'entiers. On initialise le contenu à 0 par exemple, car tous les C_i^j auront une valeur supérieure ou égale à 1. On stocke ensuite les résultats au fur et à mesure qu'ils sont trouvés. La place mémoire utilisée est $O((n-p)p) = O(np)$, mais le gain en temps d'exécution est énorme ($O(np)$ au lieu d'exponentiel !).

Passage à un algorithme itératif Cherchons un bon ordonnancement des tâches sur le graphe de dépendances. L'idée consiste à faire les calculs par ligne, en remontant dans le graphe, ce qui fait qu'**on a seulement besoin de stocker une ligne** (tableau $1..p$) avant de calculer la ligne au dessus (voir la figure 5). En d'autres termes, pour calculer C_n^p :

1. calculer la ligne d'en dessous jusqu'à C_{n-1}^p, stocker les résultats ;
2. calculer la queue de la ligne courante, de C_{n-p}^0 en remontant jusqu'à C_{n-1}^{p-1} (on peut écraser au fur et à mesure les valeurs stockées dans le début du tableau) ;
3. sommer les résultats obtenus pour C_{n-1}^p (case p du tableau) et pour C_{n-1}^{p-1} (case $p-1$ du tableau).

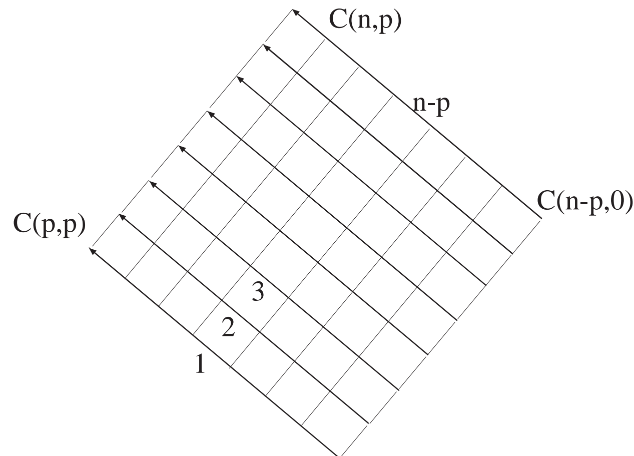


Figure 5: Un ordonnancement des tâches dans le graphe de dépendances des C_n^p.

Cet algorithme s'écrit simplement de manière itérative :

```
public int c(int n, int p) {
    // Le cache est un tableau unidimensionnel
    // Initialisation : pour tout i, cache[i] = C(i,i) = 1
    for (i = 1; i <= p; i++)
        setCache(i, 1) ;

    for (k = 1; k <= n-p; k++)
        // Calcul de la ligne debutant a C(k,p)
        for (i = 1; i <= p; i++)
            setCache(i, getCache(i) + getCache(i-1)) ;
            // Utilise le fait que cache[0] = C(k,0) = 1,
            // et que cache contenait la ligne du dessous
    return getCache(p);
}
```

Le temps d'exécution est le même que l'algorithme récursif avec cache ($O(np)$), en revanche on a gagné en place mémoire : $O(p)$ contre $O(np)$.

3.2.3 Exercice corrigé : un problème associatif

Supposons que vous êtes responsables d'une association (Cercle, BDE Entreprise, N'Sigma, ou autre), pour laquelle vous êtes en charge de l'organisation d'un évènement (soirée, gala, . . .). Il s'agit d'un évènement important et complexe à organiser, pour lequel vous avez plusieurs (n) *prestations* à gérer :

- location de la salle ;
- sonorisation ;
- boissons ;
- sécurité ;
- etc.

Vous avez lancé un appel d'offres, à laquelle deux entreprises ont répondu : Nasa Music (entreprise 1) et Bouquetin d'Argent (entreprise 2). Chaque entreprise vous propose soit de prendre en charge l'intégralité des prestations, soit une partie seulement, mais dans ce cas avec des frais supplémentaires de déplacement. En effet, on suppose que ces prestations doivent être réglées *séquentiellement*, et faire déplacer une entreprise plusieurs fois engendre un coût supplémentaire.

La figure 6 résume le graphe des coûts :

- $p_{i,j}$ correspond au coût de la prestation j si c'est l'entreprise i qui s'en charge ;
- $d_{i,j}$ correspond au coût de déplacement de l'entreprise i au début de la prestation j ;
- il existe, pour chacune des deux entreprises, un coût final fixe f_i (pour l'état des lieux, etc.).

Si vous choisissez une seule entreprise i pour l'ensemble des prestations, il vous en coûtera $d_{i,1} + (\sum_{j=0}^{n} p_{i,j}) + f_i$. Mais vous avez le droit de panacher.

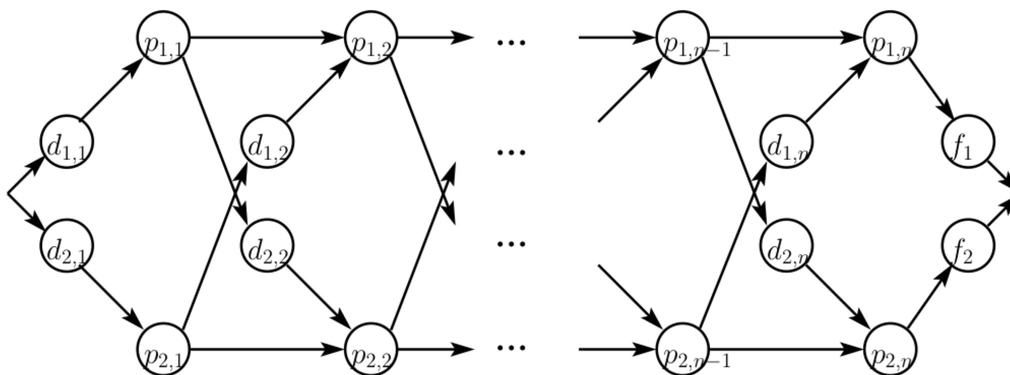


Figure 6: Graphe des coûts.

Etant donné que votre association n'est pas très riche (les subventions ont été maigres cette année), quelle solution choisissez-vous ?

<ins>Exemple numérique</ins> : $n = 6, f_1 = 3, f_2 = 2$, et

j	1	2	3	4	5	6
$p_{1,j}$	7	9	3	4	8	4
$d_{1,j}$	2	2	1	2	2	1
$p_{2,j}$	8	5	6	4	5	7
$d_{2,j}$	4	2	3	1	3	4

Analyse théorique

Question 2 *Quelle est la complexité (en fonction de n) d'un algorithme exhaustif, qui teste toutes les combinaisons possibles ?*

Corrigé 2 $\Omega(2^n)$, *puisqu'on a n choix de prestations à faire, et à chaque fois 2 possibilités.*

Question 3 *Soit C^* le coût minimum total. Soit, $\forall 1 \leq j \leq n, C_i(j)$ le coût minimum pour l'ensemble des prestations de 1 à j, en considérant que c'est l'entreprise i qui fait la prestation numéro j. Dit autrement, dans le graphe de la figure 6 (dont on notera que seuls les nœuds sont pondérés, pas les arcs), $C_i(j)$ représente le coût minimum de tous les chemins allant du "départ" au nœud pondéré par $p_{i,j}$. C^* représente le coût minimum de tous les chemins allant du "départ" à l'arrivée.*
Exprimer C^* en fonction de $C_1(n)$ et $C_2(n)$.

Corrigé 3 $C^* = \min(C_1(n) + f_1, C_2(n) + f_2)$ *: soit on a terminé avec l'entreprise 1 (i.e. cette entreprise a fait la dernière prestation), et l'état des lieux final a lieu avec cette entreprise, soit c'est avec l'entreprise 2.*

Question 4 *Soit $2 \leq j \leq n$, et soit $1 \leq i \leq 2$. Exprimer $C_i(j)$ en fonction de $C_i(j-1)$ et de $C_k(j-1)$, avec $k = 3 - i$ (c'est-à-dire que $k = 2$ si $i = 1$, et $k = 1$ si $i = 2$).*

Corrigé 4 $C_i(j) = \min\left[(C_i(j-1), C_k(j-1) + d_{i,j}\right] + p_{i,j}$ *: soit on conserve la même entreprise et il n'y a pas de coût supplémentaire, hormis le coût de la nouvelle prestation $p_{i,j}$ avec cette entreprise, soit on change d'entreprise et il faut ajouter le coût $d_{i,j}$ de déplacement de la nouvelle entreprise.*

Question 5 *Que valent les $C_i(1)$?*

Corrigé 5 *Trivialement, $C_i(1) = d_{i,1} + p_{i,1}$, car on a une seule prestation. Il ne faut pas oublier le coût de déplacement de l'entreprise.*

Le problème posé est maintenant modélisé sous forme de formule récursive. Nous avons également fixé les conditions initiales. Résolvons ce problème par programmation dynamique. Remarquons que le graphe de la figure 6 correspont au graphe de dépendances de la fonction récursive !

Question 6 *Donner la complexité en pire cas d'un algorithme récursif naïf de calcul de C^*, exploitant la récurrence sur les $C_i(j)$ trouvée ci-dessus.*

Corrigé 6 *Comme l'algorithme glouton, les calculs redondants nous donnent une complexité exponentielle : $O(2^n)$ (dessiner le graphe d'appels pour s'en rendre bien compte). Remarquer que si on a k entreprises, la récurrence devient $C_i(j) = \min_{k \neq i}(\dots) + p_{i,j}$, ce qui donne une complexité en $O(k^n)$.*

Question 7 *Donner la complexité en pire cas d'un algorithme récursif avec marquage, i.e. mémoire cache, et la place mémoire nécessaire.*

Corrigé 7 *On marque tous les $C_i(j)$ pour éviter les calculs redondants : en pratique on utilisera un tableau bidimensionnel de taille $2 \times n$. La place mémoire requise vaut donc $2n$ (kn dans le cas général). Pour la complexité, on peut appliquer le raisonnement suivant :*

- *on calcule chaque $C_i(j)$ exactement une fois ;*

- *le coût de calcul d'un $C_i(j)$ est de 2 (k dans le cas général) ;*
- *il y a $2n$ $C_i(j)$ (kn dans le cas général) ;*
- *à la fin, on fait à nouveau un min sur 2 éléments.*

La complexité est donc de l'ordre de $2 * 2n + 2$ ($k * kn + k$ dans le cas général), soit en $O(n)$ ($O(k^2n)$ dans le cas général).

L'écriture de l'algorithme récursif avec marquage est laissée à titre d'exercice personnel.

Question 8 *Trouver un ordre topologique du graphe des appels permettant de calculer C^* (indice : ce graphe ressemble fortement à celui de la figure 6). En déduire le principe d'un algorithme itératif, et donner sa complexité en pire cas et la place mémoire nécessaire.*

Corrigé 8 *On parcourt les $C_i(j)$ j après j : d'abord $C_1(1)$ et $C_2(1)$, puis $C_1(2)$ et $C_2(2)$, etc. On ne stocke que deux de ces "lignes", soit $2k$ valeurs dans le cas général : place mémoire en $O(k)$, i.e. constante dans notre cas. La complexité en pire cas ne change pas.*

L'écriture de l'algorithme itératif est laissée à titre d'exercice personnel.

Question 9 *Donner le principe d'un algorithme itératif qui calcule non seulement la valeur de C^*, mais également la solution optimale permettant d'atteindre cette valeur (autrement dit, qui explicite à quelle entreprise est affectée chaque prestation). Donner sa complexité en pire cas et la place mémoire nécessaire.*

Corrigé 9 *Il faut stocker, pour chaque $C_i(j)$, la valeur parmi 1 et 2 qui réalise le min. On ajoute donc un tableau de $2n$ booléens (ou kn entiers dans le cas général). La place mémoire nécessaire redevient donc linéaire par rapport à n, mais le reste ne change pas.*

La modification de l'algorithme itératif est laissée à titre d'exercice personnel.