École Polytechnique Promotion 2000

Corrigé du contrôle hors-classement d'Informatique Fondamentale

Georges Gonthier

23 Avril 2002

Cet examen est composé de deux parties indépendantes. Tous les documents du cours sont autorisés. Le correcteur attachera une grande importance à la concision la clarté, et la précision de la rédaction.

Partie I : Modélisation de scènes 3D

L'objet de cette partie est l'écriture d'une mini-bibliothèque de rendu de scènes 3D, limitée à un lancer de rayon simpliste sur des scènes polyhédriques. On identifiera les points à des vecteurs de $\mathbb{R}^3$, et on les représentera par des objets de la classe `Vec3` ci-dessous ; ainsi, le produit scalaire de deux vecteurs `v1` et `v2` s'écrira `v1.prod(v2)`.

```
class Vec3 {
  float x, y, z;
  Vec3 (float x0, float y0, float z0) { x = x0; y = y0; z = z0; }
  float prod (Vec3 v) { return x * v.x + y * v.y + z * v.z; }
  final static Vec3 NUL = new Vec3 (0, 0, 0);
}
```

Un rayon, c'est-à-dire une demi-droite ouverte, sera représenté par deux vecteurs $\overrightarrow{org}$ et $\overrightarrow{dir}$ donnant respectivement l'origine et la direction de la demi-droite (la norme de $\overrightarrow{dir}$ importe peu).

Question 1. Écrivez la classe `Rayon` correspondante, avec des champs `org` et `dir`, en y incluant un constructeur et une méthode `pos(float` λ`)` qui retourne le point de paramètre $\lambda > 0$ du rayon, $\overrightarrow{pos}(\lambda) = \overrightarrow{org} + \lambda\overrightarrow{dir}$.

Corrigé.

```
class Rayon {
  Vec3 org, dir;
  Rayon (Vec3 o, Vec3 d) { org = o; dir = d; }
  Vec3 pos (float a) {
    return new Vec3 (org.x + a * dir.x, org.y + a * dir.y, org.z + a * dir.z);
  }
}
```

□

La scène à rendre sera assimilée à une partie fermée de $\mathbb{R}^3$, et modélisée par une combinaison de primitives géométriques à l'aide des opérations ensemblistes binaires union et intersection. On se limite à une seule primitive graphique, le demi-espace, et une seule méthode de rendu, le `lancer(Rayon` r`)`, qui retourne le `Trajet` du rayon, c'est-à-dire la liste des demi-espaces correspondant aux surfaces de la scène qui sont traversées par le rayon, dans l'ordre des λ croissants.

Un demi-espace est défini par une normale $\overrightarrow{norm}$ et une «distance» à l'origine $\overline{dist}$ (qui peut être négative) : c'est l'ensemble des points $\vec{p}$ tels que $\overrightarrow{norm}.\vec{p} \leq \overline{dist}$.

Question 2. Écrivez les définitions des classes `Scene`, `Union`, `Inter`, `DemiEspace`, et `Trajet` correspondant à cette architecture, en utilisant l'héritage, et en incluant les constructeurs, mais en omettant le corps de `lancer(Rayon` r`)`, qui fera l'objet d'une question ultérieure.

Corrigé.

```
abstract class Scene {
  abstract Trajet lancer (Rayon);
}
class Union extends Scene {
  Scene sg, sd;
  Union (Scene s1, Scene s2) { sg = s1; sd = s2; }
  Trajet lancer (Rayon r) { ... }
}
class Inter extends Scene {
  Scene sg, sd;
  Inter (Scene s1, Scene s2) { sg = s1; sd = s2; }
  Trajet lancer (Rayon r) { ... }
}
class DemiEspace extends Scene {
  Vec3 norm; float dist.
  DemiEsp (Vec3 n, float d) { norm = n; dist = d; }
  Trajet lancer (Rayon r) { ... }
}
class Trajet {
  DemiEspace face; Trajet suivant;
  Trajet (DemiEspace f, Trajet t) { face = f; suivant = t; }
}
```

□

Question 3. Écrivez une méthode `pave(Vec3 p, Vec3 q)` qui retourne la `Scene` constituée du parallépipède rectangle dont les côtés sont parallèles aux axes et dont `p` et `q` sont deux sommets diagonalement opposés. (Si `q` est dans l'octant supérieur de `p`, cette `Scene` sera l'ensemble des points de coordonnées (x, y, z) telles que `p.x` $\leq x \leq$ `q.x` et `p.y` $\leq y \leq$ `q.y` et `p.z` $\leq z \leq$ `q.z`.)

Corrigé.

```
static Scene pave (Vec3 p, Vec3 q) {
  float dx = p.x - q.x, dy = p.y - q.y, dz = p.z - q.z;
  Scene fpx = new DemiEspace (new Vec3 (-dx, 0, 0), -dx * p.x);
  Scene fpy = new DemiEspace (new Vec3 (0, -dy, 0), -dy * p.y);
  Scene fpz = new DemiEspace (new Vec3 (0, 0, -dz), -dz * p.z);
  Scene fqx = new DemiEspace (new Vec3 (dx, 0, 0), dx * q.x);
  Scene fqy = new DemiEspace (new Vec3 (0, dy, 0), dy * q.y);
  Scene fqz = new DemiEspace (new Vec3 (0, 0, dz), dz * q.z);
  Scene cxy = new Inter (new Inter (fpx, fqx), new Inter (fpy, fqy));
  return new Inter (cxy, new Inter (fpz, fqz));
}
```

□

Question 4. Ajoutez une méthode sans arguments `compl()` à `Scene` pour calculer la scène complémentaire (plus précisément, le complémentaire de l'intérieur de la scène, c'est-à-dire que le complémentaire d'un demi-espace est un demi-espace), et définissez-la dans toutes les sous-classes de `Scene` (vous pouvez ignorer les cas dégénérés de demi-espaces tangents). Utilisez `compl()` pour définir une méthode `diff(Scene` s_1`, Scene` s_2`)` qui calcule la différence entre deux scènes.

Corrigé.

```
// dans Scene
abstract Scene compl();
 // dans Union
Scene compl () { return new Inter (sg.compl(), sd.compl()); }
// dans Inter
Scene compl () { return new Union (sg.compl(), sd.compl()); }
// dans DemiEsp
Scene compl () {
  return new DemiEsp (new Vec3 (-norm.x, -norm.y, -norm.z), -dist);
}
```

□

Un rayon peut soit entrer, soit sortir, soit être disjoint de, soit être inclus dans ou tangent à un demi-espace. On ajoute donc les constantes suivantes à la classe `Rayon`.

```
final static int ENTRE = 0, SORT = 1, DISJOINT = 2, INCLUS = 3;
```

Question 5. Ajoutez à la classe `Rayon` une méthode `incidence(DemiEsp` f`)` qui retourne l'un des quatre codes ci-dessus, ainsi qu'une méthode `parametre(DemiEsp` f`)` qui calcule le $\lambda > 0$ pour lequel f`.pos(`λ`)` est sur la surface du demi-espace, dans le cas où `incidence(`f`)` retourne `ENTRE` ou `SORT`.

Corrigé.

```
int incidence (DemiEsp f) {
  float a = f.dist - f.norm.prod(org), b = f.norm.prod(dir);
  if (a * b > 0) return a < 0 ? ENTRE : SORT;
  return a - b < 0 ? DISJOINT : INCLUS;
}
float parametre (DemiEsp f) {
  return (f.dist - f.norm.prod(org)) / f.norm.prod(dir);
}
```

□

Afin de simplifier le lancer de rayon et de permettre de déterminer une unique couleur d'affichage, on s'impose de ne produire que des trajets *alternés*, où les surfaces apparaissent par λ *strictement* croissant, et les entrées alternent avec les sorties ; en particulier on supprimera les intersections d'épaisseur nulle.

En outre, un trajet *plein* qui ne comporte pas d'intersections parce qu'il est entièrement inclus dans la scène sera représenté non pas par la liste vide, mais par la constante

```
final static Trajet PLEIN = new Trajet (new DemiEsp (Vec3.NUL, 1), null);
```

Question 6. Ajoutez à la classe `Rayon` une méthode `alterne(Trajet` t`)` qui teste que le trajet est bien alterné pour le rayon.

Corrigé.

```
boolean alterne (Trajet t) {
  if (t == null || t == Trajet.PLEIN) return true;
  int i = incidence (t.face); float a = 0;
  while (t != null) {
    if (incidence (t.face) != i) return false;
    float at = parametre (t.face);
    if (at <= a) return false;
    i = (i == SORT ? ENTRE : SORT); a = at; t = t.suivant;
  }
  return true;
}
```

□

Question 7. Definissez les méthodes `lancer(Rayon` r`)` des sous-classes de `Scene`. On vous recommande d'utiliser une ou plusieurs méthodes auxiliaires récursives dans la classe `Rayon` pour fusionner les trajets des opérations `Inter` et `Union` (il est possible de factoriser ces deux cas). On ne précise pas de règle de priorité lorsqu'un rayon coupe plusieurs faces en un même point.

Corrigé.

```
// dans DemiEspace
Trajet lancer (Rayon r) {
  int i = r.incidence(this);
  if (i == Rayon.DISJOINT) return null;
  if (i == Rayon.INCLUS) return Trajet.PLEIN;
  return new Trajet (this, null);
}
// dans Inter
Trajet lancer (Rayon r) {
  return r.fusion (sg.lancer(r), sd.lancer(r), true);
}
// dans Union
Trajet lancer (Rayon r) {
  return r.fusion (sg.lancer(r), sd.lancer(r), false);
}
// dans Rayon, méthodes auxiliaire et auxiliaire récursive
Trajet fusion (Trajet t1, Trajet t2, boolean inter) {
  if (t1 == null || t2 == Trajet.PLEIN) return inter ? t1 : t2;
  if (t2 == null || t1 == Trajet.PLEIN) return inter ? t2 : t1;
  int i1 = incidence (t1), i2 = incidence (t2);
  boolean m = (i1 == SORT) ^ inter;
  return fusion (t1, t2, m, i1 != i2, parametre (t1.face));
}
// * m  == le Trajet t1 commence par un segment masquant la fusion
//         (c'est-a-dire plein pour une union, vide pour une intersection).
// * d  == les Trajets t1 et t2 commencent par des type de segment différents.
// * a1 == parametre (t1.face).
Trajet fusion (Trajet t1, Trajet t2, boolean m, boolean d, float a1) {
  if (t2 == null) return (m ^ d) ? t2 : t1;
  float a2 = parametre (t2.face);
  if (m ? a1 < a2 : a1 <= a2) { // echanger t1 et t2
```

```
    Trajet t = t1; t1 = t2; t2 = t; m ^= d; a1 = a2;
  } // ici on a parametre(t2) <= a1, strictement si m est faux et d est vrai
  Trajet t = fusion (t1, t2.suivant, m, !d, a1);
  return m ? t : new Trajet (t2.face, t);
}
```

□

Partie II : Recherche de cliques dans un graphe non orienté

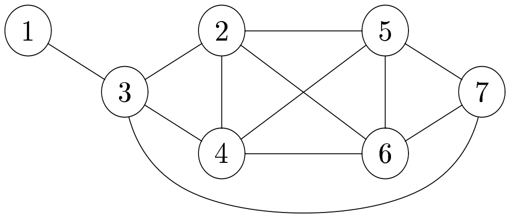
Dans cette partie on se place dans un graphe non orienté $G = (V, E)$. Une *clique* de G est une partie X de V qui induit un sous-graphe complet de G et qui est maximale pour cette propriété, c'est-à-dire que

1. $X \subseteq V$
2. Si $x, y \in X$, avec $x \neq y$, alors $(x, y) \in E$
3. Si $x \in V \setminus X$, alors il existe un $y \in X$ tel que $(x, y) \notin E$.

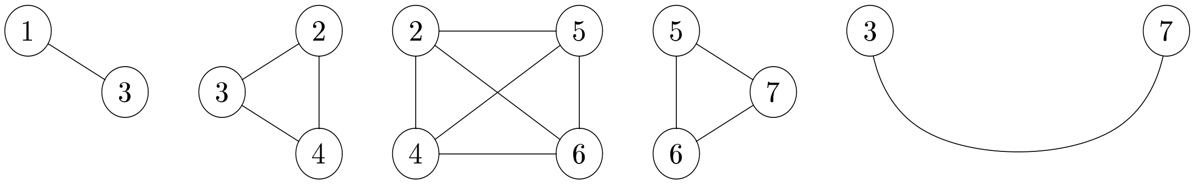
La notion de clique a plusieurs applications, notamment en analyse de données (data-mining), car il permet d'identifier des groupes homogènes (p. ex., une communauté de pages Web).

Le problème ci-dessous porte sur l'énumération de toutes les cliques de G. Pour les questions de programmation, on utilisera la représentation par liste d'adjacence du cours (classes `Liste` et `Graphe`).

Question 8. Listez toutes les cliques du graphe suivant :



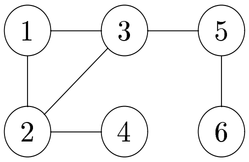
Corrigé. On a les cinq cliques suivantes.



□

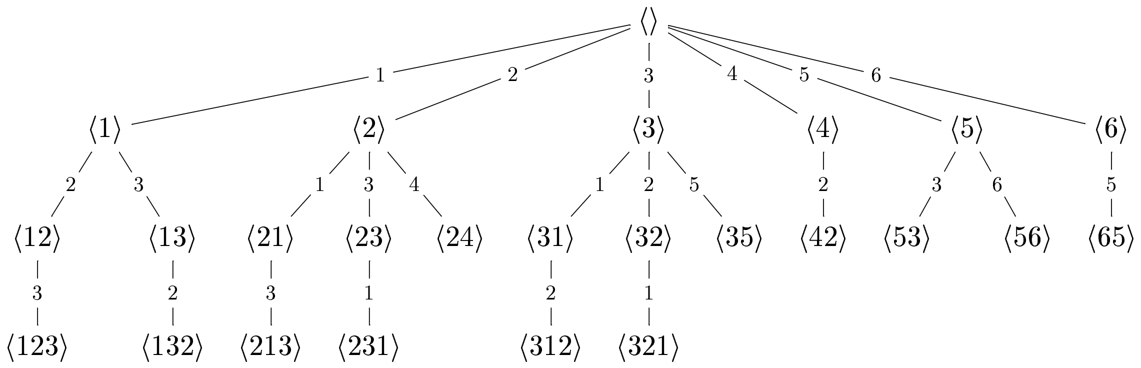
Considérons l'ensemble $\mathcal{X}$ de toutes les séquences $\sigma = x_0 \cdots x_{k-1}$ de sommets distincts de V qui induisent un sous-graphe complet de G (i.e., $x_i \neq x_j$ et $(x_i, x_j) \in E$ pour tout $i \neq j$, $0 \leq i < k$, $0 \leq j < k$). $\mathcal{X}$ est un arbre (il est clos par préfixe), et ses feuilles correspondent aux cliques de G.

Question 9. Dessiner l'arbre $\mathcal{X}$ pour le graphe G suivant :



Combien de fois apparaissent dans $\mathcal{X}$ chacune des cliques de G ?

Corrigé.



La clique $\{1,2,3\}$ apparaît 6 fois, les cliques $\{2,4\}$, $\{3,5\}$ et $\{5,6\}$ apparaissent 2 fois chacunes. □

Question 10. Ajoutez à la classe `Graphe` une méthode `cliques()` qui imprime toutes les cliques de G en effectuant un parcours préfixe naïf de l'arbre $\mathcal{X}$. Utilisez une `Pile` s pour stocker la séquence σ correspondant au nœud courant, que vous pourrez imprimer à l'aide de l'instruction `System.out.println(`s`)`. (On vous suggère d'utiliser une méthode récursive auxiliaire, et de stocker dans un tableau `ns`, pour chaque sommet x le nombre de sommets de σ qui sont voisins de x.)

Corrigé.

```
void cliques () {
  int n = succ.length; Pile s = new Pile(n); int[] ns = new int[n];
  for (int x = 0; x < n; x++) cliques (x, s, 0, ns);
}
void cliques (int x, Pile s, int h, int[] ns) {
  for (Liste a = succ[x]; a != null; a = a.suivant) ++ns[a.val];
  boolean feuille = true; s.empiler(x);
  for (Liste a = succ[x]; a != null; a = a.suivant)
    if (ns[a.val] > h) { feuille = false; cliques (a.val, s, h + 1; ns); }
  for (Liste a = succ[x]; a != null; a = a.suivant) --ns[a.val];
  if (feuille) System.out.println(s);
  s.depiler();
}
```

□