Structures de Données - Rapport de conception.

Dubuc Xavier & Hannecart Aurore

15 février 2010

Table des matières

---

1 Introduction

Dans le cadre du cours de *Structures de données II*, nous avons été amenés à mener à bien un projet concernant les arbres **BSP***(Binary Space Partition)*. Ces arbres sont des arbres binaires de recherche spéciaux permettant de partitionner une scène constituée de segments. Notre projet consiste à afficher ce que voit un œil d'une scène en 2 dimensions sur une droite en utilisant l'**algorithme du peintre**. Pour ce faire, nous devons construire un arbre **BSP** selon une heuristique fournie et interroger celui-ci.

2 Classes utilisées

2.1 Couleur.java

Cette classe est une classe plus utile que nécessaire, en effet, elle permet simplement de faire la correspondance entre une chaîne de caractères et un objet *Color*. Ainsi, on crée des objets contenant par exemple «Bleu» et *Color.BLUE* et on implémente des méthodes utiles comme *equals()* et *toString()* afin de faciliter la correspondance décrite ci-dessus.

2.2 Palette.java

Cette classe *abstraite* représente simplement la palette de couleurs que l'on utilise dans ce projet. Elle contient un tableau d'objet *Couleur* et implémente quelques méthodes de classe permettant d'obtenir l'objet *Couleur*, l'objet *Color* ou la chaine de caractères caractérisant une couleur en fonction d'une chaine de caractères d'un objet *Couleur* ou d'un objet *Color*.

2.3 EquationDroite.java

Cette interface permet de symboliser l'équation d'une droite, elle demande comme méthode *eval(double x, double y)*. Elle nous servira pour calculer $ax + by + c$ pour un point désiré.

2.4 Segment.java

Cette classe permet de symboliser un segment, un segment étant caractérisé par 2 points (classe java *Point2D.Double*), une couleur (classe *Couleur*) et l'équation de la droite qui le porte (interface *EquationDroite*). On y implémente la méthode *equals()* de manière à ce que 2 segments soient égaux si ils sont de même couleur et ont les 2 mêmes points les définissant. On implémente également une méthode *hasIntersectionWith()* ainsi que *getIntersectionPointWith()* qui prennent toutes 2 une **EquationDroite** en paramètre. La première permet de savoir si la droite portant le segment possède un et un seul point d'intersection avec la droite passée en paramètre et la seconde permet de calculer ce point.

2.5 SceneReader.java

Cette classe permet de lire un fichier *.txt* représentant une scène et de convertir les informations en un Arraylist de segments tout en conservant également le nombre de segments, l'abscisse maximale et l'ordonnée maximale.

2.6 BSPTree.java

Cette classe représente la structure de données des arbres **BSP**, nous avons décidé d'utiliser une seule classe, classe pouvant représenter une feuille contenant un seul segment, vide ou un noeud ayant 1 ou 2 fils. Elle possède un ArrayList contenant tous les segments qui sont contenus dans le séparateur, séparateur représenté par une **EquationDroite**. Elle possède également 2 booléens spécifiant si le noeud est une feuille et si celui-ci est vide ainsi que 2 **BSPTree** symbolisant les 2 fils du noeud. (Ces 2 fils pouvant ne pas exister)

2.7 Heuristic.java

Interface permettant de définir un heuristique. Elle demande la méthode *getDroite(ArrayList<Segments> a)*, méthode qui est appellée à chaque séparation effectuée. Celle-ci permet donc de définir quelle droite

utiliser pour séparer la scène. Comme exemple simple, l'heuristique consistant à choisir la droite portant le premier segment de l'ensemble consiste à simplement retourner $a.get(0)$.

2.8 BSPTreeBuilder.java

Cette classe *abstraite* permet de construire un arbre **BSP** en fonction d'une heuristique prédéfinie. Elle consiste en l'application de l'algorithme vu dans le livre en permettant de spécifier l'heuristique à utiliser, cette classe est donc totalement générale et pourra être utilisée pour toutes les heuristiques que l'on aura à implémenter.

2.9 Vue.java

Définie par 2 points, un point de situation et un point d'atteinte. Le premier est le point où se situe l'œil et le second permet de définir dans quel sens regarde cet œil.

2.10 DrawPanel.java

Classe étendant la classe **JPanel** de java et qui permet de dessiner le segment vu par un œil défini.

2.11 Painter.java

Cette classe abstraite permet d'appliquer l'algorithme du peintre et d'afficher la réponse demandée par l'énoncé, c'est-à-dire le segment représentant ce que voit la **Vue** via un **DrawPanel**. Elle possède 3 méthodes importantes,

- *getDepthOrder(BSPTree tree, Vue v)* qui renvoie un ensemble de segments triés par ordre d'apparition inversé face à l'oeil (le dernier segment de la liste est le premier devant l'œil)
- *getProjectedSegment(Vue v, Segment a)* qui renvoie un **Segment** correspondant à la projection de a sur la droite perpendiculaire au vecteur définissant l'œil.
- *getViewedSegment(Vue v, ArrayList a)* qui appelle la méthode précédente pour chaque segment de l'ensemble de segments triés par ordre d'apparition afin de dessiner une droite représentant ce que voit l'oeil. Il reste ensuite à convertir les coordonnées afin que la droite soit horizontale et bien centrée pour l'afficher sans problème.

Cette manière de faire n'est pas très optimale car on parcoure 3 fois l'arbre ou la liste des segments pour effectuer un travail que l'on peut faire en parcourant une seule fois l'arbre (en se collant complètement à l'algorithme explicité dans le livre). Nous avons cependant décidé dans un premier temps de tout séparer afin que le code soit plus clair et plus compréhensible pour nous, lors de l'implémentation, le code sera optimisé.

3 Algorithmes

3.1 Algorithme 1 : Création d'un arbre BSP

3.2 Algorithme 2 : Heuristique

Rappel de l'heuristique : heuristique numéro 4, l'heuristique de *Teller* ; l'idée est de choisir la droite d qui :

- **maximise** σ_d *(la proportion de segments coupés par d)* si σ_d est $\geq$ qu'un certain seuil donné représenté par un nombre réel,
- **minimise** f_d *(le nombre de segments coupés par d)* sinon.

3.3 Algorithme 3 : algorithme du peintre

L'idée ici est de parcourir l'arbre selon un ordre défini et de travailler sur chaque segment considéré. Le travail consiste à projeter le segment sur une droite derrière la scène, de vérifier s'il n'est pas caché et de le dessiner sur la droite (avec sa couleur). Comme nous avons séparé le travail, l'algorithme du peintre tel que nous l'avons utilisé est très léger et simple, il nous permet uniquement de récupérer les segments dans l'ordre dans lequel l'œil les voit. Voici donc cet algorithme, qui est pratiquement identique à celui du livre.

Algorithm 1 BuildBSPTree(S,h)

Entrée : S, *une liste de segment de taille n représentant une scène,* h, *l'heuristique à suivre pour construire l'arbre.*

Sortie : T, *l'arbre BSP correspondant.*

if $(n = 0)$ **then**
  $T_{separator} \leftarrow$ vide
  $T_{left} \leftarrow$ vide
  $T_{right} \leftarrow$ vide
  $T_{segments} \leftarrow$ vide
  $T_{isLeaf} \leftarrow$ vrai
  $T_{isEmpty} \leftarrow$ vrai
else
  if $(n = 1)$ **then**
    $T_{separator} \leftarrow S[0]_{equation}$
    $T_{left} \leftarrow$ vide
    $T_{right} \leftarrow$ vide
    $T_{isLeaf} \leftarrow$ vrai
    $T_{isEmpty} \leftarrow$ faux
    $Ajout(S[0], T_{segments})$
  else
    $T_{separator} \leftarrow$ *GetDroite(S,h)*
    $T_{isLeaf} \leftarrow$ faux
    $T_{isEmpty} \leftarrow$ faux
    $S^+ \leftarrow nouvelleListe()$
    $S^- \leftarrow nouvelleListe()$
    for i allant de 0 à $n-1$ **do**
      s $\leftarrow$ S[i]
      $valBegin \leftarrow Evaluation(T_{separator}, s_{begin})$
      $valEnd \leftarrow Evaluation(T_{separator}, s_{end})$
      if $(valBegin = 0$ ET $valEnd = 0)$ **then**
        $Ajout(s, T_{segment})$
        if $(valBegin \leq 0$ ET $valEnd \leq 0)$ **then**
          $Ajout(s, S^-)$
        else
          if $(valBegin \geq 0$ ET $valEnd \geq 0)$ **then**
            $Ajout(s, S^+)$
          else
            $m \leftarrow Intersection(T_{separator}, s)$
            $s_1 \leftarrow (s_{begin}, m, s_{color})$
            $s_2 \leftarrow (m, s_{end}, s_{color})$
            if $(valBegin \leq 0)$ **then**
              $Ajout(s_1, S^-)$
              $Ajout(s_2, S^+)$
            else
              $Ajout(s_1, S^+)$
              $Ajout(s_2, S^-)$
            end if
          end if
        end if
      end if
    end for
    $T_{left} \leftarrow$ **BuildBSPTree**(S^-, h)
    $T_{right} \leftarrow$ **BuildBSPTree**(S^+, h)
    retourner T
  end if
end if

Algorithm 2 GetDroite(I_v,τ)

Entrée : I_v, *une liste de segment de taille n représentant une scène,* τ *un nombre réel représentant le seuil.*

Sortie : d, *l'équation de la droite à utiliser comme séparateur.*

$(max_{\sigma_d}, segmentMax_{\sigma_d}) \leftarrow (0, (+\infty, +\infty))$
$(min_{F_d}, segmentMin_{F_d}) \leftarrow (+\infty, (+\infty, +\infty))$
for all les segments s de I_v **do**
  $d \leftarrow$ *droite supportant* s
  $f_d \leftarrow 0$
  $\sigma_d \leftarrow 0$
  for all les segments s' de I_v (sauf s) **do**
    if d *intersecte* s*'* **then**
      $f_d \leftarrow f_d + 1$
    end if
  end for
  $\sigma_d \leftarrow \dfrac{f_d}{|I_v|}$
  if $\sigma_d \geq max_{\sigma_d}$ **then**
    $(max_{\sigma_d}, segmentMax_{\sigma_d}) \leftarrow (\sigma_d, s)$
  end if
  if $f_d \geq min_{f_d}$ **then**
    $(min_{f_d}, segmentMin_{f_d}) \leftarrow (f_d, s)$
  end if
end for
if $\sigma_d \geq \tau$ **then**
  retourner *la droite portant* $segmentMax_{\sigma_d}$
else
  retourner *la droite portant* $segmentMin_{f_d}$
end if

Algorithm 3 GetDepthOrder(T,v)

Entrée : T, *un arbre BSP,* v *une vue (telle que décrite plus haut).*
Sortie : A, *un arraylist contenant les segments dans l'ordre décrit ci- haut.*

```
if T est vide then
  retourner arraylist vide
else
  if T est une feuille then
    Ajout(A, T_segments[0])
    retourner A
  else
    A ← arraylist vide
    d ← T_separator
    val ← Evaluation(d, v)
    Right ← GetDepthOrder(T_right, v)
    Left ← GetDepthOrder(T_left, v)
    if val = 0 then
      for all segments s dans Right do
        Ajout(A, s)
      end for
      for all segments s dans Left do
        Ajout(A, s)
      end for
    else
      if val ≤ 0 then
        for all segments s dans Right do
          Ajout(A, s)
        end for
        for all segments s dans T_segment do
          Ajout(A, s)
        end for
        for all segments s dans Left do
          Ajout(A, s)
        end for
      else
        for all segments s dans Left do
          Ajout(A, s)
        end for
        for all segments s dans T_segment do
          Ajout(A, s)
        end for
        for all segments s dans Right do
          Ajout(A, s)
        end for
      end if
    end if
  end if
  retourner A
end if
```