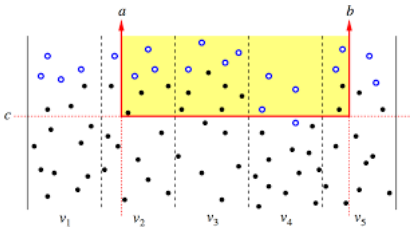
U-MONS

Rapport de projet de Structures de données 2

Priority Search Tree and Windowing



Directeurs :
G.Devillez et V.Bruyère

Groupe :
M.Salemi et A.Lecocq

17 avril 2017

Table des matières

1 Introduction

Dans le cadre du cours de structures de données 2, nous avons été amenés à réaliser un projet en Java. Ce projet a pour objectif de créer et manipuler une structure de données non vue au cours. Cette nouvelle structure se base sur un arbre de recherche à priorité (Priority Search Tree en anglais ou PST). De la documentation nous a été fournie afin de nous familiariser avec cette dernière structure qui elle non plus n'a pas été vue au cours.

Un PST partage certaines caractéristiques avec les tas et les Arbres Binaires de Recherche ou ABR. Ces deux dernières structures de données ayant été étudiées en profondeur lors des séances de cours, nous les considérons ici comme connues.

2 Présentation du problème

Il nous a été demandé d'écrire un programme qui permette d'appliquer un "windowing" à un ensemble de segments de droite verticaux et horizontaux dans $\mathbb{R}^2$. Le windowing est une technique qui consiste à extraire tous les segments qui sont visibles dans une fenêtre donnée. Les différentes fenêtres à implémenter sont les suivantes :

- $[X, X']$ x $[Y, Y']$;
- $[-\infty, X']$ x $[Y, Y']$;
- $[X, +\infty]$ x $[Y, Y']$;
- $[X, X']$ x $[-\infty, Y']$;
- $[X, X']$ x $[Y, +\infty]$.

La structure de données à implémenter devrait donc permettre un windowing efficace pour l'ensemble de ses fenêtres.

3 Priority Search Tree

3.1 Objectif de la structure

Un PST est une structure de données de type arbre binaire (chaque nœud comporte au plus deux fils). Cette structure de données organise des points de l'espace défini par deux coordonnées X et Y. L'organisation des données permet d'effectuer efficacement la recherche des points présents dans une fenêtre de l'espace (sans avoir à parcourir l'ensemble des points).

3.2 Définition

Un PST est une structure de données mixte. Chaque nœud est constitué d'un point et d'un nombre appelé la médiane. Si l'on considère uniquement les coordonnées X, un PST est un tas avec le minimum à la racine. Si l'on considère uniquement les coordonnées Y, le PST est un ABR avec une petite particularité. Au lieu de trier les fils par rapport à la donnée du nœud courant, un PST trie les fils en fonction de la médiane du nœud courant.

Ainsi, tout nœud n d'un PST respecte les contraintes suivantes :
- son fils gauche (s'il existe ainsi que ses descendants s'ils existent) aura sa coordonnée X plus grande que celle du nœud n et sa coordonnée Y plus petite que la médiane du nœud n ;
- son fils droit (s'il existe ainsi que ses descendants s'ils existent) aura sa coordonnée X plus grande que celle du nœud n et sa coordonnée Y plus grande que la médiane du nœud n.

3.3 Construction

3.3.1 Tri des données

La construction d'un PST est plus simple si l'on construit ce dernier à partir d'une liste de points triés selon la coordonnée Y. La première étape est donc de trier les points.

3.3.2 Création de l'arbre

Pour créer un arbre, créons le nœud racine à partir de la liste des points triés.

3.3.3 Création d'un nœud

La coordonnée X d'un nœud devant être plus petite ou égale à celle de ses fils, commençons par rechercher le point avec la plus petite coordonnée X. Attribuons ce point au nœud courant. Séparons le reste des points en deux. Attribuons comme médiane du nœud courant, la moyenne de la coordonnée Y du dernier point de la partie 1 et du premier point de la partie 2. Les points étant triés selon la coordonnée Y, la première partie est dispose d'une coordonnée Y plus petite ou égale à la médiane du nœud alors que la deuxième partie aura une coordonnée Y plus grande ou égale à la médiane du nœud. Dans le cas où il ne reste aucun point après le retrait du minimum en X, la

valeur de la moyenne n'a pas d'importance. Dans le cas où il ne reste qu'un point, la coordonnée Y du point restant peut être attribuée à la médiane du nœud.

3.4 Windowing

Le windowing est une technique très répandue qui consiste à sélectionner une certaine fenêtre parmis une énorme quantité de données. Un exemple pratique très répandu est l'affichage d'une carte sur un gps, le gps se voulant rapide n'affichera pas toutes les routes qui se trouvent dans le monde(énorme quantité de données) mais seulement celles qui nous entourent au moment où nous roulons avec notre véhicule.

4 PST Adapté

Étant donné que notre structure s'applique à des segments et non à des points, nous ne pouvons utiliser le PST tel que décrit ci-dessus. Nous avons donc adapté cette structure afin de permettre l'utilisation de segments.

Nous avons donc considérer deux façons d'adapté le problème à la structure initiale.
La première étant de considérer les deux points définissant un segment séparément, tout en gardant un lien vers le segment de base.
La deuxième approche est de considérer les segments et adapter les différents algorithmes de la structure de base à des segments dans le plan.
Nous avons après réflexion décider d'utiliser la deuxième façon d'adapté la structure.

Un segment est défini par 2 points, chacun ayant une coordonnée X et une coordonnée Y. Un segment possède donc 2 coordonnées X ainsi que 2 coordonnées Y. Un segment est soit horizontal soit vertical, c'est-à-dire soit les coordonnées X soit les coordonnées Y sont égales.
Afin de garder les explications concises et faciles à comprendre, nous allons d'abord considérer uniquement une fenêtre de type $[X, X']$ x $[Y, Y']$ ou $[-\infty, X']$ x $[Y, Y']$ et nous expliquerons ensuite comment gérer les autres cas.

4.1 Objectif de la structure

L'objectif de cette structure est d'organiser l'ensemble des segments afin de faciliter la recherche de segments visibles dans une fenêtre de type $[X, X']$ x $[Y, Y']$ ou $[-\infty, X']$ x $[Y, Y']$.

4.2 Définition

Notre PST adapté respecte exactement la définition d'un PST excepté 2 adaptation :
- un segment sera stocké par nœud au lieu d'un point ;
- les coordonnées utilisées pour effectuer la comparaison sont le minimum des coordonnées X du segment comme X, le minimum des coordonnées Y du segment comme Y.

Le choix des coordonnées de référence est judicieux car il permet lors de la recherche d'une fenêtre, d'élaguer tout nœud (ainsi que ses descendants par définition du tas) dont la coordonnée minimum en X est supérieure à X' et il permet de ne pas visiter le fils droit (ainsi que ses descendants par définition de l'ABR) de tout nœud dont la médiane est strictement supérieure à Y'.

4.3 Construction

Notre PST adapté se construit à la manière d'un PST excepté que les références pour la comparaison des données sont les coordonnées minimums en X ainsi qu'en Y. C'est-à-dire que nous trions les segments selon leur plus petite coordonnée en Y. Lorsque nous recherchons le minimum en X, nous recherchons le segment avec la plus petite

coordonnée X.

L'article conseillait également lors de la construction de la structure d'utiliser une méthode bottom-up similaire à la construction d'un tas. Nous avons ici décidé de ne pas utiliser cette méthode mais de construire l'arbre de haut en bas à l'aide des segments fournis à la structure. Une explication plus détaillée est fournie lors de l'explication de l'algorithme au point 6.

4.4 Windowing

Étant donné que nous devons reporter tous les segments visible dans une certaine fenêtre, nous devons non seulement extraire les segments ayant au moins un point dans la fenêtre, mais aussi les segments commençant avant et finissant après (qui passent au travers de la fenêtre). C'est le cas notamment du segment possédant les coordonnées (X', Y-1) (X', Y'+1). La recherche sera donc un peut moins efficace que pour les points et nous allons devoir utiliser une approche différente.

Nouvelle approche : se concentrer sur le point de départ du segment c'est-à-dire le point ayant la plus petite coordonnée X pour un segment horizontal, le point ayant la plus petite coordonnée Y pour un segment vertical. Nous avons ici 3 possibilités pour que le segment soit visible :

- soit le point de départ est dans la fenêtre ;
- soit le point de départ est à gauche de la fenêtre et le segment traverse la limite gauche de la fenêtre ;
- soit le point de départ est en dessous de la fenêtre et le segment traverse la limite basse de la fenêtre.

Il est évident que si le point de départ se situe au dessus ou à droite de la fenêtre, ce segment ne sera pas visible dans la fenêtre.

Nous allons donc parcourir l'arbre à partir de la racine et le traiter de la sorte :

- si le minimum en X du segment est strictement supérieur à X', alors on laisse tomber ce nœud ainsi que tous ses descendants (le segment de ce nœud ainsi que celui de ses descendants a son point de départ à droite de la fenêtre) ;
- si ce segment est visible dans la fenêtre (voir conditions ci-dessus), le reporter ;
- si la médiane du nœud est strictement supérieure à Y' (le segment du fils droit ainsi que celui de ses descendants a son point de départ au dessus de la fenêtre), alors parcourir le fils gauche uniquement sinon parcourir les deux fils.

L'utilisation de cette structure pour réaliser un windowing de type $[-\infty, X']$ x $[Y, Y']$ est très efficace car aucun nœud inutile n'est parcouru.

4.5 Les autres fenêtres

4.5.1 De $[-\infty, X']$ x $[Y, Y']$ à $[X, X']$ x $[-\infty, Y']$

Nous avons vu une structure de données très efficace pour réaliser un windowing de type $[-\infty, X']$ x $[Y, Y']$. Cependant, nous devons également réaliser un windowing avec une fenêtre de type $[X, X']$ x $[-\infty, Y']$. Nous remarquons que pour obtenir cette fenêtre efficacement, il suffit d'échanger les coordonnées X et Y. Nous aurions pu implémenter une structure de donnée supplémentaire étant un tas selon Y et un ABR selon X (avec les médianes). Nous avons choisi de réutiliser la même structure de

données et d'inverser les coordonnées des segments avant la construction de l'arbre. Ainsi le segment (X, X'),(Y, Y') devient après échange le segment (Y, Y'),(X, X'). Nous effectuons une recherche avec la fenêtre échangée $[-\infty, Y']$ x $[X, X']$ et effectuons un échange inverse sur les segments reportés afin de retrouver les segments originaux.

4.5.2 Les autres fenêtres partiellement bornées

Maintenant que nous pouvons effectuer une recherche efficace sur des fenêtres de type $[-\infty, X']$ x $[Y, Y']$ et $[X, X']$ x $[-\infty, Y']$, il nous reste les fenêtres $[X, +\infty]$ x $[Y, Y']$et $[X, X']$ x $[Y, +\infty]$. Nous remarquons que pour obtenir ces fenêtres efficacement, il suffit d'opposer les coordonnées. Nous aurions pu ici aussi créer une structure de donnée avec une comparaison différente. Nous avons également choisi de réutiliser la même structure et d'opposer les coordonnées avant la construction de l'arbre. Ainsi, le segment (X, X'),(Y,Y') devient après opposition (-X',-X),(-Y',-Y).

4.5.3 La fenêtre fermée

Lors de la recherche dans une fenêtre fermée avec l'arbre décrit ci-dessus, nous ne pouvons jamais éliminer le fils gauche de nos recherches. En conséquence, les nœuds que nous parcourons ayant un minimum en X strictement inférieur au X de la fenêtre et ayant un minimum en Y strictement inférieur au Y de la fenêtre sont parcourus inutilement. Ces segment sont ceux qui commencent en bas à gauche de la fenêtre et il est évident qu'ils n'y seront pas visibles. Nous avons donc adapté notre windowing afin qu'il prenne deux paramètre :

- center : définit si les segments commençant au centre de la fenêtre doivent être reportés ;
- down : définit si les segments commençant en dessous de la fenêtre doivent être reportés.

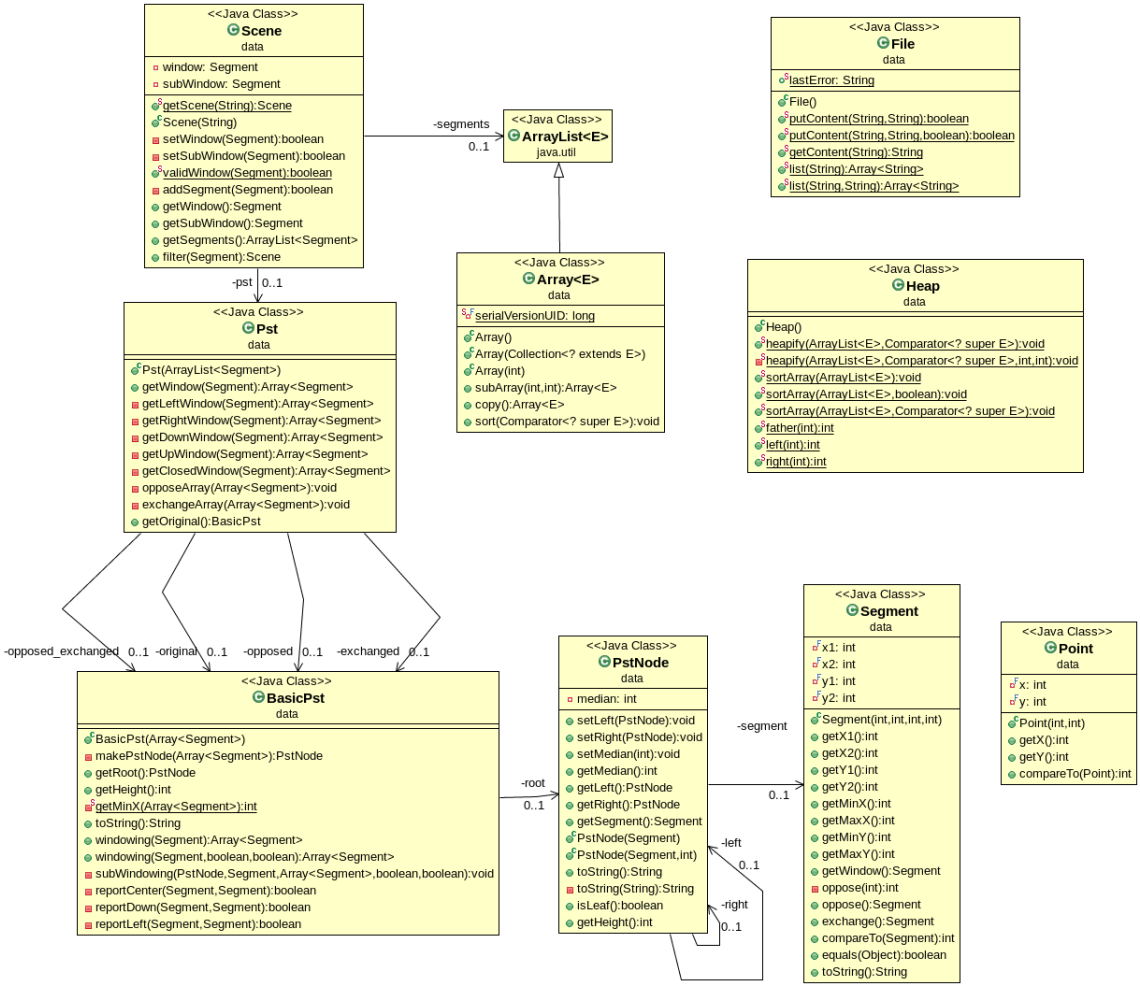
Si center est faux, alors nous pouvons laisser tomber les nœuds qui ont un minimum en X plus grand que le X de la fenêtre. Si down est faux, alors nous pouvons laisser tomber les fils gauches des nœuds dont la médiane est strictement inférieure au Y de la fenêtre.

Afin d'effectuer la recherche de la fenêtre fermée efficacement, nous appliquerons le windowing deux fois. Une première fois avec down à faux afin de reporter les segments ayant le point de départ à gauche et dans la fenêtre. Une seconde fois sur la fenêtre échangée avec la même technique que pour la fenêtre $[X, X']$ x $[-\infty, Y']$ avec center et down à faux afin de reporter tous les segments ayant leur point de départ en dessous.

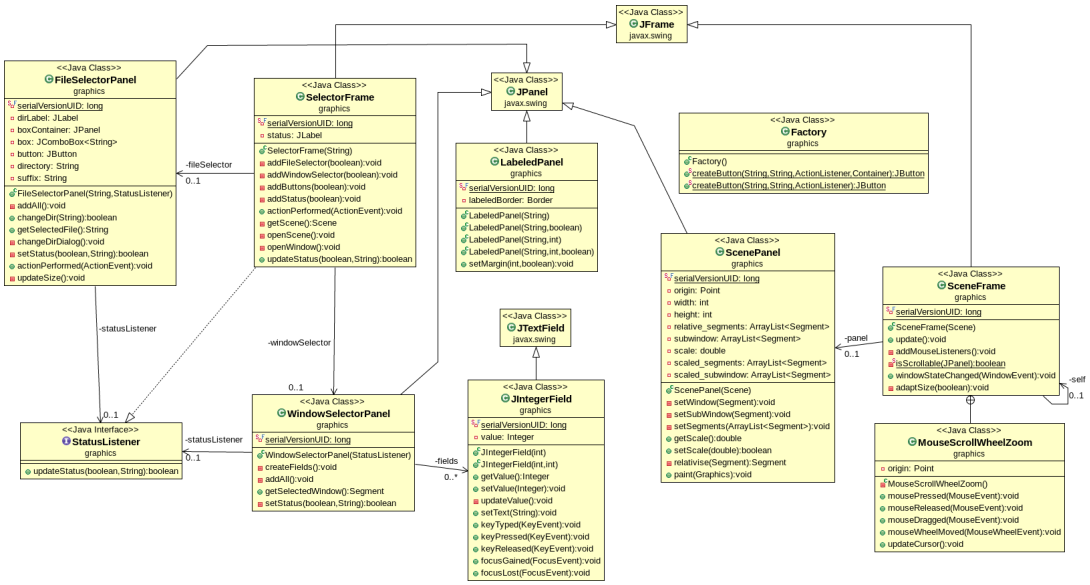
5 Diagrammes de classes

Nous vous présentons durant ce point l'entièreté de nos classes à travers deux diagrammes de classes distincts, l'un concernant le travail sur les données, et l'autre le travail sur l'interface graphique de l'application. Ils sont ensuite suivis de différents points afin de vous expliquer le but des différentes classes implémentées.

5.1 Diagramme de classe des données



5.2 Diagramme de classe de l'interface graphique



5.3 Pst et BasicPst

BasicPst correspond au PST gérant efficacement une fenêtre de type $[-\infty, X'] \text{ x } [Y, Y']$. Afin de gérer toutes les transformations, nous avons créé une autre classe Pst qui comporte 4 BasicPst :

1. un avec les segments originaux pour les fenêtres $[-\infty, X'] \text{ x } [Y, Y']$;
2. un avec les segments échangés pour les fenêtres $[X, X'] \text{ x } [-\infty, Y']$;
3. un avec les segments opposés pour les fenêtres $[X, +\infty] \text{ x } [Y, Y']$;
4. un avec les segments échangés et opposés pour les fenêtres $[X, X'] \text{ x } [Y, +\infty]$;

Cette classe s'occupe de choisir l'arbre sur lequel effectuer le windowing afin d'obtenir la meilleure efficacité.

6 Algorithmes et explications

Dans cette section, nous exposons nos différents algorithmes importants en pseudo-code ainsi que leur complexité en temps. Une courte explication en français y est aussi ajoutée afin de favoriser la compréhension de ceux-ci et leurs utilisations.

6.1 Construction de l'arbre

Algorithme 1 Construction de l'arbre

Entrée: une liste de Segment A, un PstNode $Root$ qui est la racine de l'arbre
Sortie: /

1: $A \leftarrow heapSort(A)$
2: $root \leftarrow construct(A)$

Algorithme 2 Construct()

Entrée: une liste de Segment triée en y A
Sortie: un PstNode qui représente la racine de l'arbre

1: **si** A est vide **alors**
2: **return** $null$
3: **end si**
4: $node \leftarrow A[getMinX()]$
5: **si** A est non vide **alors**
6: $node.median \leftarrow longueur(A)/2$
7: $node.left \leftarrow construct(A.sousListe(0, node.median))$
8: $node.right \leftarrow construct(A.sousListe(node.median, longueur(A)))$
9: **end si**
10: **return** node

Algorithme 3 getMinX()

Entrée: une liste de Segment L
Sortie: l'indice entier de l'élément minimum en X

1: **si** L est vide **alors**
2: **return** -1
3: **end si**
4: $min \leftarrow 0$
5: **for** $i \leftarrow 1$ à $longueur[A]$ **do**
6: **si** $A[i].getMinX() < A[min].getMinX()$ **alors**
7: $min \leftarrow i$
8: **end si**
9: **end for**
10: **return** min

6.1.1 Explication

Nous construisons ici l'arbre de haut en bas, ayant initialement la racine qui est le minimum en x des segments, nous effectuons ensuite une séparation des données en

deux vis à vis de la médiane en y, et ceci récursivement. Biensur pour que ceci soit possible, nous trions les segments en y avant d'utiliser la liste pour la création de l'arbre.

6.1.2 Complexité dans le pire des cas

Nous commençons d'abord par calculer la complexité dans le pire des cas en temps de l'algorithme 3. Les 4 premières lignes sont en $\mathcal{O}(1)$, ainsi que les lignes 7 et 10, car le retour est constant, la condition du if est constante et une affectation de valeur est constante en temps. La condition du if de la ligne 6 est elle aussi constante, car nous faisons une comparaison de valeurs (constante) et qu'avoir accès à la valeur du minimum x d'un segment se fait en temps constant. Nous aurons donc un for en ligne 5 qui sera en $\mathcal{O}(n)$ où n est la taille de la liste. L'algorithme possède donc une complexité de $\mathcal{O}(n)$ au total.

Pour ce qui concerne l'algorithme 2, les lignes se font toutes en temps constant pour les mêmes raisons que celles énoncées pour le 3e algorithme mise à part pour les lignes 4,7 et 8. En effet pour les 7,8 nous effectuons un appel récursif qui sera exécuter en $\mathcal{O}jesaispas$. Et pour la ligne 4, le getMinX() se résouds en $\mathcal{O}(n)$ par le point développer ci-dessus.

Pour se qui concerne l'algorithme 1, celui ci possède une première ligne en $\mathcal{O}(n.log(n))$ (complexité en temps du heapSort), et une seconde en $\mathcal{O}(n^2)$. Nous aurons donc une complexité finale de $\mathcal{O}(n^2)$

6.2 Windowing

6.3 Modification des fenêtres

7 Mode d'emploi

Pour lancer une commande, rendez-vous dans le dossier racine du projet à l'aide du terminal.

7.1 Javadoc

Pour compiler la Javadoc, entrez la commande `ant javadoc`. La documentation compilée se trouve alors dans le répertoire **build/doc/**.

7.2 Tests unitaires

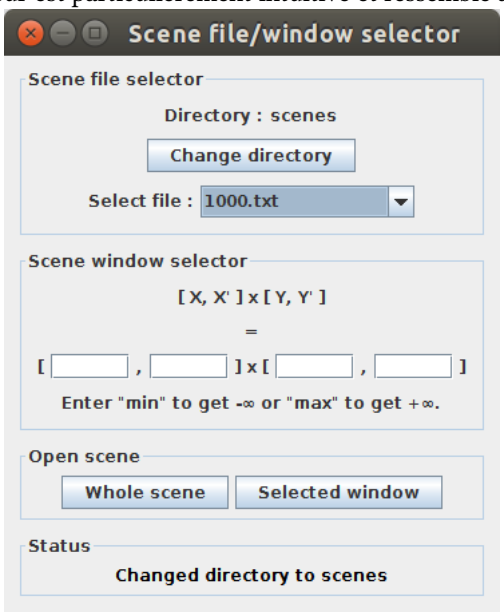
Pour lancer les tests unitaires, entrez la commande `ant test`.

7.3 Lancement du programme

Pour compiler et lancer le programme à partir des fichiers sources, entrez la commande `ant run`.

7.4 Interface utilisateur

L'interface utilisateur est particulièrement intuitive et ressemble à ceci :



Comme vous pouvez le voir, l'interface est divisée en 4 parties :

7.4.1 Sélecteur de fichier

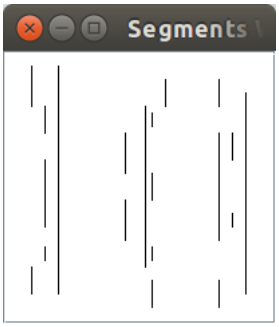
La première partie contient un sélecteur de fichier qui vous proposera par défaut de choisir un fichier .txt parmi ceux présents dans le dossier **scenes** (relatif à l'application). Vous pouvez changer de répertoire si vous souhaitez choisir des fichiers qui se trouvent à un autre emplacement.

7.4.2 Sélecteur de fenêtre

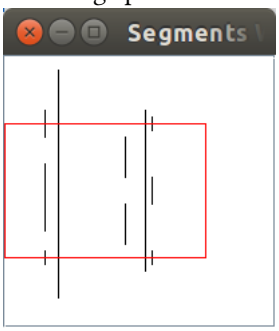
La seconde partie vous permet d'indiquer au programme la fenêtre à utiliser pour effectuer le windowing. Entrez "min" et "max" pour indiquer respectivement $-\infty$ et $+\infty$.

7.4.3 Ouvreur de scène

La troisième partie vous propose deux boutons. L'un ouvrira toute la scène contenue dans le fichier sélectionné par le sélecteur de fichier dans une nouvelle fenêtre. Exemple :



L'autre bouton ouvrira la scène contenue dans le fichier sélectionné après y avoir appliqué le windowing en utilisant la fenêtre fournie par le sélecteur de fenêtre. La fenêtre sélectionnée s'affichera en rouge par dessus la scène. Exemple :



Si la scène est trop grande pour entrer dans la fenêtre de visualisation nouvellement ouverte, des barres de défilement apparaîtront et il vous sera possible de naviguer dans la scène en maintenant le pointeur de la souris enfoncé.

Si vous le désirez, il est possible de changer la taille de la visualisation à l'aide de la molette de votre souris.

7.4.4 Information sur l'état du programme

La quatrième et dernière partie de l'interface affiche l'état du programme. Les erreurs y sont affichées en rouge alors que les informations y sont en noir.

8 Illustrations

Ce point du rapport sert d'illustration, vous permettant d'observer différents cas d'exemples d'application du windowing, et ceci via l'application que nous avons créer.

9 Conclusion

Nous avons bien réalisé les objectifs fixés dans l'introduction à savoir créer et manipuler une structure de données non vue au cours.
Il est clair que ce travail nous a demandé à plusieurs reprises de réfléchir sur des moyens algorithmiques de résoudre les problèmes rencontrés. Nous avons donc appliquer les différents concepts et structures vus durant le cours de SDD2 afin de trouver des solutions et savoir quelles structures sont les mieux adaptées et comment les appliquées.
Ce travail nous aura donc permis de finalement faire "un peu de recherche" en groupe, dans l'optique de résoudre un problème nouveau, ce qui nous a permis de découvrir lors de nos réunions de groupe nos différentes idées, de les comparer et de considérer le problème sous différents aspects. Nous aurons donc un souvenir très positif de se travail.
Nous tenons à remercier les personnes qui nous ont soutenues pour mener à bien ce projet à savoir BRUYÈRE Véronique et DEVILLEZ Gauvain.