# Partitionnement de l'espace et calcul d'un rendu graphique à l'aide d'un arbre BSP et de l'algorithme du peintre

## Structures de Données 2
## Rapport de projet

Nathan Amorison
Emilien Lemaire

15/08/2022


# Introduction

L'affichage d'une projection d'une scène efficacement et rapidement, peu importe la dimension, est une problématique récurrente des outils graphiques largement répandus dans le domaine informatique.

Pour ce faire, partitionner la scène et créer un arbre reprenant des sous-espaces et leurs informations (arbre BSP, que nous définirons plus en détail dans les premières pages) est une solution intéressante. Pour ce qui est de l'affichage en tant que tel, l'algorithme du peintre, dont nous expliquerons également le fonctionnement dans ce rapport dans la partie dédiée aux algorithmes, permet de définir efficacement, grâce à l'arbre BSP, quels sont les éléments appartenant au premier, second et dernier plan, depuis un point de vue. Cette stratégie permet ainsi l'affichage de ce que pourrait voir un oeil dans la scène en fonction de la direction de son regard ainsi que de son champ de vision (Nous parlerons tout au long ce rapport de FOV, Field Of View, pour y faire référence).

Dans ce projet, nous avons donc implémenté l'arbre BSP et l'algorithme du peintre, ainsi que différentes heuristiques permettant de diversifier les façons de diviser l'espace, afin de représenter des projections 1D de ce que voit un oeil placé dans des scènes en 2D constituées de segments non sécants. Nous avons également comparé ces différentes heuristiques et leur impact sur l'efficacité des résultats calculés par la stratégie décrite ci-dessus.

# Table des matières

# 1 Structures de Données et heuristiques utilisées

Nous présentons ici les différentes structures de données que nous avons considérées comme étant intéressantes/importantes à implémenter ainsi qu'une description de leur implémentation. Certaines structures de données ont été imaginées dès les premières phases de l'analyse de la problématique tandis que d'autres ont été implémentées selon les besoins pendant les différentes phases d'implémentation.

1. **Point2D**
   Un point2D est défini par ses coordonnées x et y sur son plan Vectoriel.

2. **Segments**
   Un segment est une droite visible sur la scène, il est défini par sa couleur, 2 Point2D qui le délimite ainsi qu'un booléen par point afin de savoir si le Point2D se trouve sur le segment d'un de ses parents dans l'arbre.

   a. *Données*
      - *firstPoint*, *lastPoint*

        Ces données sont des *Point2D* permettant de stocker les coordonnées d'origine et de fin d'un segment.
      - *firstOnEdge*, *lastOnEdge*

        Booléens permettant de savoir si le point correspondant se trouve sur la droite contenant un autre segment dans la scène.
      - *color*

        Permet de stocker la couleur associée à un segment. La classe EColor, un enum, permet la transcription entre les couleur sous format "textuel", tel que dans les fichiers scènes, en couleur utilisable par javafx.

   b. *Méthodes*
      - *set*

        Permet de paramètrer les coordonnées *x1*, *y1*, *x2*, *y2* du segment.
      - *setColor*

        Permet de paramétrer la couleur *color* du segment.
      - *getFrom*

Récupère le point d'origine du segment.

— *getTo*

Récupère le point de fin du segment.

— *getEColor*

Récupère la couleur *color* du segment.

— *isFreeSplit*

Permet de savoir si le segment est un FreeSplit [1].

c. *Fonctions*

— *cut*

Permet de récupérer les deux morceaux d'un segment passé en argument lors d'une découpe par un autre segment passé en argument.

— *getCutlineParameters*

Permet de récupérer les paramètres $a$, $b$, $c$ de l'équation d'une droite, dans un espace à deux dimensions, tel qu'elle contienne le segment passé en argument.

— *getIntersection*

Récupère les coordonnées de l'intersection entre deux droites grâce à leurs paramètres donnés en argument.

— *getAngle*

Récupère l'angle entre un segment et un point. Permet de définir l'angle entre la direction de la vue et un point dans la scène.

— *getProjection*

Récupère la coordonnée $x$ en une dimension de la projection d'un point, depuis le point de vue de l'oeil, sur le canvas d'écriture de la solution.

d. *Implémentation de IVector*

— *getDistance*

Récupère la distance cartésienne du segment.

— *getX*, *getY*

1. Un segment FreeSplit est un segment dont les extrémités se trouvent sur les bornes d'un espace. Le segment coupe alors de part-et-d'autre cet espace. Nous décrivons son utilité dans la description de l'heuristique du FreeSplit au point 5 de cette section.

Récupère les composantes $x$ et $y$ du vecteur associé au segment.

— *isMultipleOf*

Vérifie que 2 segment, sous forme de vecteurs, sont multiples l'un de l'autre.

— *isNull*

Vérifie si le segment, sous forme de vecteur, n'est pas nul.

*hasSameSens*

Vérifie si 2 vecteurs ont le même sens.

— *getFactor*

Récupère le facteur entre deux segments s'ils sont multiples l'un de l'autre, sous forme de vecteurs.

— *getPerp*

Récupère un des segments perpendiculaires.

3. **Eye**

L'oeil est l'emplacement choisi par l'utilisateur comme point de vue de la scène, il est défini par ses coordonnées x y, par sa direction et par son angle.

4. **Arbre BSP**

Un arbre BSP (Binary Space Partition) est une structure de données disposée en arbre binaire. Cet arbre est construit en divisant récursivement l'espace en 2 parties dont le nœud correspond à la ligne de découpe tandis que ses 2 fils contiennent chacun une moitié de l'espace coupé par cette ligne. Ce processus se termine lorsqu'il ne reste plus qu'un seul élément dans un sous ensemble. Dans notre situation, la ligne de découpe est obtenue par la droite passant par un segment de la scène. Ainsi, une feuille de l'arbre BSP permet de définir 2 sous-espaces vides de la scène ; il n'y a donc plus de segments dans ceux-ci. Les arbres BSP sont notamment utilisés pour l'affichage des jeux vidéos et plus généralement pour les rendus graphiques tels que la 3D.

a. *Données*

— *data*

Liste chaînée permettant de stocker des données dans le noeud racine de l'arbre.

— *left*, *right*, *parent*

Références vers les sous-arbres respectivement gauche et droit ainsi que le noeud parent.

b. *Méthodes*

— *getData*

Récupère la liste chaînée de données stockée dans le noeud.

— *getLeft*, *getRight*, *getParent*

Récupèrent respectivement les références vers les sous-arbres gauche et droit et le noeud parent.

— *isEmpty*

Vérifie si l'arbre est vide.

— *isLeaf*

Vérifie si l'arbre est en réalité une feuille.

— *height*, *length*

Récupèrent respectivement la hauteur et la taille de l'arbre.

— *setLeft*, *setRight*, *setParent*

Paramétrage respectivement des références vers les sous-arbres gauche et droit et le noeud parent.

— *setData*

Paramétrage de la liste chaînée des données stockée dans le noeud racine.

— *addData*

Ajout d'une donnée dans la liste chaînée des données stockée dans le noeud racine.

c. *fonctions*

— *makeTree*

Crée un arbre BSP.

— *paintersAlgorithm*

Algorithme du peintre. Il permet d'identifier efficacement l'ordre dans lequel les segments doivent être affichés. En effet, les segments à l'arrière plan doivent être affichés en premier tandis que les segments à l'avant plan doivent être affichés en dernier, afin de recouvrir les précédents. Cet algorithme se base sur la position de l'oeil ainsi que ses paramètres de vision (FOV, Direction).

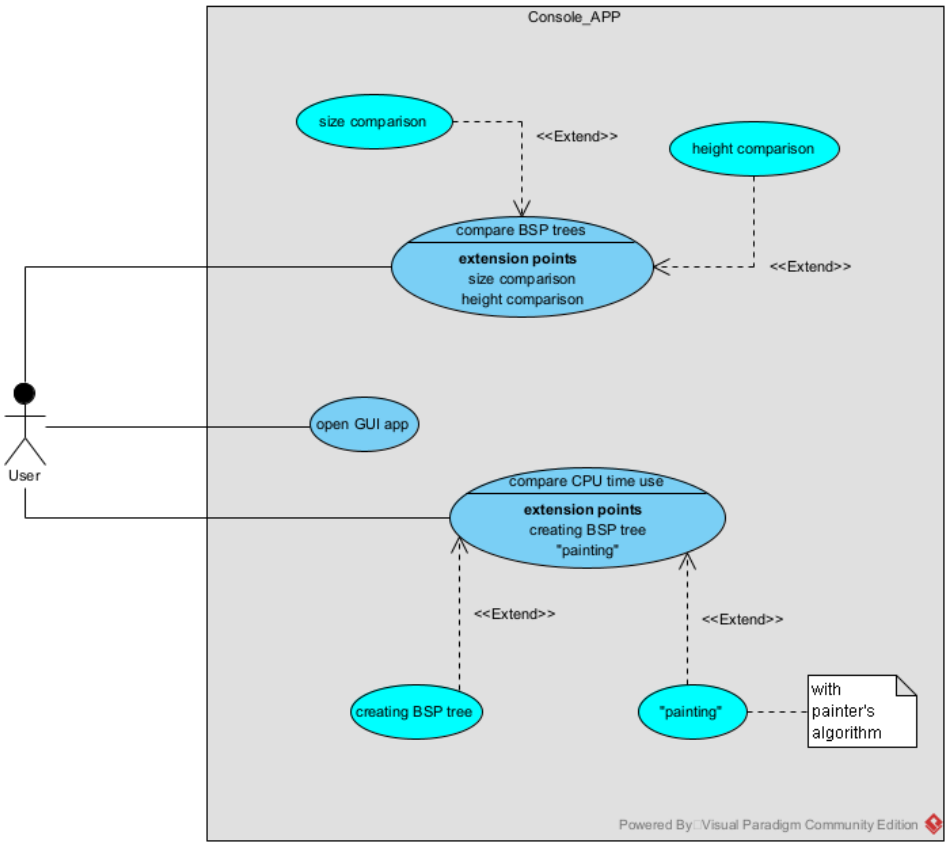
5. **Free Split**

Le Free Split est une des heuristiques implémentées dans ce projet. Cette heuristique va utiliser les segments qui traversent un sous ensemble, déjà délimité par une partie de la construction de l'arbre, de part en part. Puisque l'on sait que les segments ne sont pas sécant, aucun segment ne devra être coupé en deux. On peut directement créer les fils et rappeler l'algorithme récursivement. On peut rendre le procédé de recherche de segment étant un free split plus efficace en ajoutant un boolean par extrémité, qui sera vrai si le point est sur une ligne de coupe et faux sinon. Dès qu'un segement est coupé alors on sait que le point d'intersection sera sur la ligne de coupe et les 2 nouveaux segments auront le boolean de ce point à vrai. Lorsque les 2 boolean d'un segment sont à vrai, alors ce segment est un free split et l'on va donc échanger ce segment avec le premier de la liste et appeler la méthode de création de l'arbre bsp.
Cette heuristique tend en fait à optimiser (minimiser) le nombre de segments, après que certains aient été coupés, et donc le nombre de noeuds constituant l'arbre BSP.

# 2 Diagrammes UML

## 2.1 Use Case Diagrams

Afin de bien identifier les problématiques et d'avoir une idée claire de ce que nous avions à développer, nous avons commencé par établir des diagrammes de cas d'utilisation, ce qui nous a permis de savoir par où commencer et de bien structurer nos idées.




1. **open GUI app**

   **Nom du cas d'utilisation :** open GUI app

   **Acteur :** User

   **Description :** L'utilisateur peut lancer l'application graphique depuis l'application console.

**Hypothèses :**

— Le système affiche l'application graphique.

**Préconditions :** None.

**Déroulement basique :**

**I.** Le système affiche un menu avec les différents choix que l'utilisateur peut faire.

**II.** L'utilisateur peut choisir l'index correspondant à l'utilisation de l'application graphique.

**III.** Le système lance l'application graphique.

**Déroulement alternatif :** None.

**Postconditions :** None.

2. **compare BSP trees**

**Nom du cas d'utilisation :** compare BSP trees

**Acteur :** User

**Description :** L'utilisateur peut comparer 2 arbres BSP construits avec 2 heuristiques différentes.

**Hypothèses :**

— Le système affiche les possibilités de comparaison.

**Préconditions :** Avoir choisi un fichier scène et 2 heuristiques.

**Déroulement basique :**

**I.** Le système affiche un menu avec les différents choix que l'utilisateur peut faire.

**II.** L'utilisateur peut choisir l'index correspondant à la comparaison de la hauteur entre 2 arbres.

**Déroulement alternatif :**

**IIb.** L'utilisateur peut choisir l'index correspondant à la comparaison de la longueur entre 2 arbres.

**Postconditions :** None.

3. **compare CPU time use**

   **Nom du cas d'utilisation :** compare CPU time use

   **Acteur :** User

   **Description :** L'utilisateur peut comparer le temps CPU utilisé pour créer chacun des arbres ou pour faire fonctionner l'algorithme du peintre.

   **Hypothèses :**

   — Le système affiche les possibilités de comparaison.

   **Préconditions :** Avoir choisi un fichier scène et 2 heuristiques.

   **Déroulement basique :**

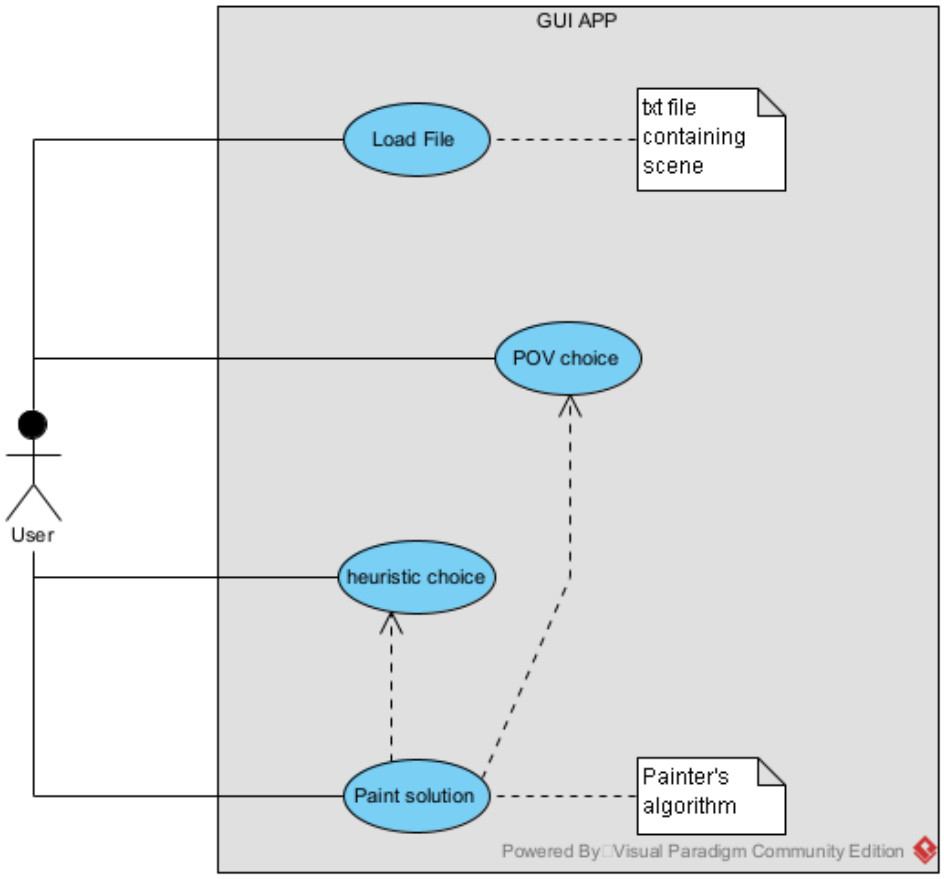
   **I.** Le système affiche un menu avec les différents choix que l'utilisateur peut faire.

   **II.** L'utilisateur peut choisir l'index correspondant à la comparaison du temps CPU utilisé pour créer les 2 arbres.

   **Déroulement alternatif :**

   **IIb.** L'utilisateur peut choisir l'index correspondant à la comparaison du temps CPU utilisé pour que l'algorithme du peintre se termine.

   **Postconditions :** None.




1. **Load File**

   **Nom du cas d'utilisation :** Load File

   **Acteur :** User

   **Description :** L'utilisateur peut ouvrir un fichier scène pour en voir la représentation.

   **Hypothèses :**

   — L'utilisateur demande au système d'ouvrir un fichier grâce au navigateur de fichier.

   **Préconditions :** None.

   **Déroulement basique :**

   **I.** Le système affiche un navigateur de fichier.

   **II.** L'utilisateur peut choisir un fichier texte.

   **III.** Le système ouvre le fichier et affiche la scène.

**Déroulement alternatif :** None.

**Postconditions :** None.

2. **POV choice**

   **Nom du cas d'utilisation :** POV choice

   **Acteur :** User

   **Description :** L'utilisateur peut choisir l'emplacement de l'oeil dans la scène ainsi que ses paramètres de direction et d'angle de vision.

   **Hypothèses :**

   — L'utilisateur choisit les différents paramètres de l'oeil dans la scène.

   **Préconditions :** L'utilisateur doit avoir chargé un fichier scène au préalable.

   **Déroulement basique :**

   **I.** L'utilisateur choisit la position de l'oeil dans la scène grâce à sa souris.

   **II.** L'utilisateur choisit la direction vers laquelle l'oeil regarde.

   **III.** L'utilisateur choisit l'angle de vision de l'oeil.

   **IV.** L'utilisateur choisit d'afficher ces paramètres sur la scène.

   **V.** Le système affiche sur la scène, en plus de l'oeil, des segments représentants les paramètres choisis.

   **Déroulement alternatif :**

   **IVb.** L'utilisateur ne choisit pas d'afficher les paramètres sur la scène.

   **Vb.** Le système n'affiche pas les paramètres en plus de l'oeil dans la scène.

   **Postconditions :** None.

3. **heuristic choice**

   **Nom du cas d'utilisation :** heuristic choice

   **Acteur :** User

**Description :** L'utilisateur peut choisir une heuristique qui sera utilisée pour calculer la solution de ce que "voit" l'oeil.

**Hypothèses :**

— L'utilisateur choisit une des 3 heuristiques.

**Préconditions :** None.

**Déroulement basique :**

**I.** L'utilisateur choisit une heuristique.

**Déroulement alternatif :** None.

**Postconditions :** None.

4. **Paint solution**

**Nom du cas d'utilisation :** Paint solution

**Acteur :** User

**Description :** L'utilisateur demande au système d'afficher la solution de ce que "voit" l'oeil.

**Hypothèses :**

— Le système récupère toutes les informations que l'utilisateur lui a fournies et calcule la solution à partir de ces dernières.

**Préconditions :** L'utilisateur doit avoir chargé un fichier scène au préalable.

**Déroulement basique :**

**I.** L'utilisateur demande l'affichage de la solution.

**II.** Le système récupère l'ensemble des segments constituant la scène, tous les paramètres relatifs à l'oeil et le choix de l'heuristique.

**III.** Le système crée l'arbre BSP.

**IV.** Le système utilise l'algorithme du peintre pour récupérer les segments dans l'ordre dans lequel ils doivent être affichés.

**V.** Le système calcule la projection de chacun de ces segments grâce aux paramètres de l'oeil et affiche la solution.

**Déroulement alternatif :** None.

**Postconditions :** None.

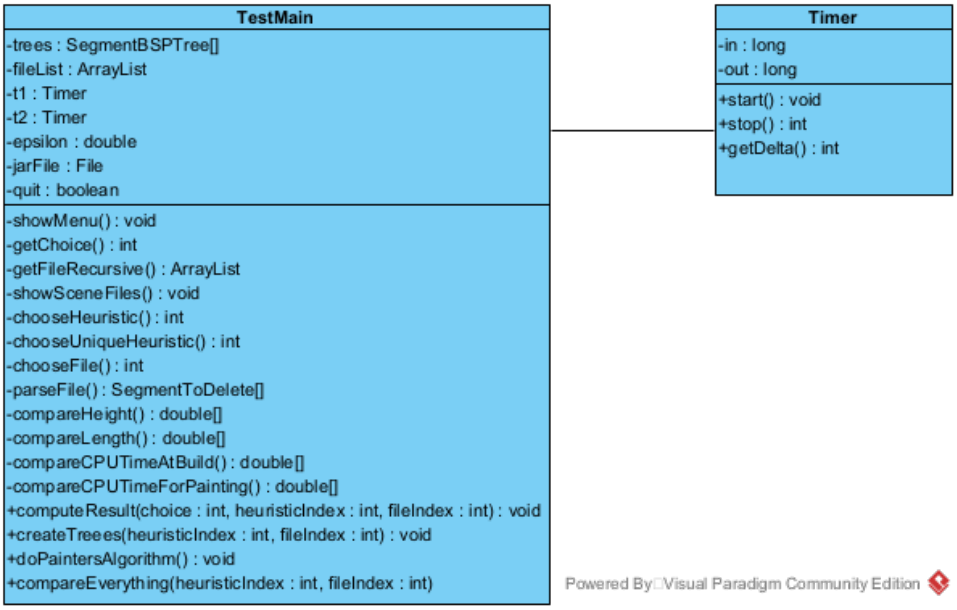
## 2.2 Class Diagrams

Nous avons divisé notre applications en 3 packages.

1. application console,
2. application graphique,
3. partie commune aux deux applications.

La partie commune aux deux applications est en réalité l'ensemble des structures de données utilisées dans l'application (Principalement les segments et les arbres BSP).

En ce qui concerne l'application graphique, nous avons opté pour une architecture MVC (Model View Controller). A ce titre, les vues ne sont pas des classes java mais sont contenues dans des fichiers fxml disponibles dans le dossier des ressources du projet gradle. Afin de distinguer les fichiers relatifs aux contrôleurs et aux modèles, ils sont chacun contenus dans deux sous-packages distinctifs.



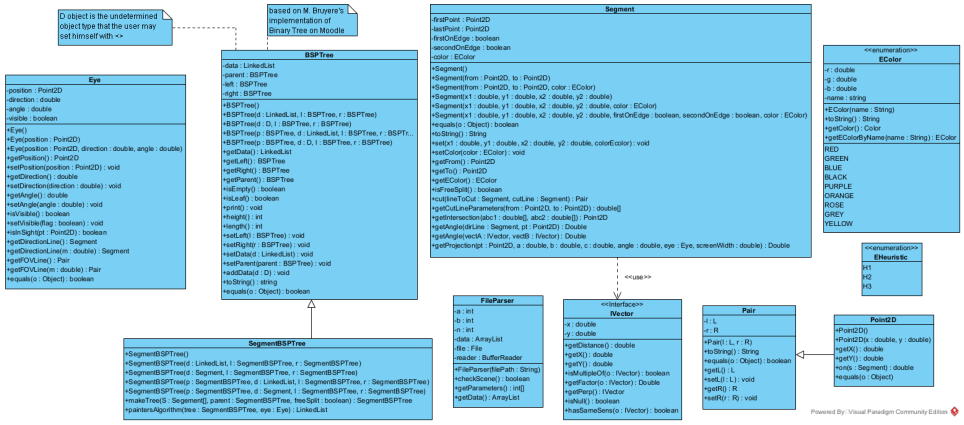

1. **TestMain**

— Il s'agit de la classe principale de l'application où se trouve le main principal. Il s'agit en réalité de l'application console dans laquelle se retrouve

donc les méthodes importantes telles que *compareHeight()*, *compareLength()*, *compareCPUTimeAtBuild()* et *compareCPUTimeForPainting()* qui sont les implémentations principales de l'énoncé du projet concernant l'application console.

2. **Timer**

— *Timer* est une classe qui permet de calculer des delta de temps en nanosecondes afin de calculer les temps CPU utilisés, pour les comparaisons.




1. **EHeuristic**

— Enumération représentant les 3 heuristiques.

2. **EColor**

— Enumération complexe représentant les couleurs utilisées pour les segments.

3. **IVector**

— Interface représentant un vecteur 2D utilisée dans la classe segment afin de voir les segments comme des vecteurs et les manipuler comme tels. Par exemple, vérifier la direction et le sens pour les segments représentant les paramètres de l'oeil (FOV, Direction) permettant notamment de savoir quels sont les segments visibles dans la scène.

4. **Segment**

— Plus importante structure de donnée du projet ; il s'agit d'une représentation d'un segment, implémentant l'interface IVector décrite ci-dessus.

Elle implémente des méthodes inhérentes à l'objet segment mais aussi des fonctions utiles en relation avec les objets segments, tel que *makeBasicTree* qui permet de créer un arbre BSP "simple" ou bien *paintersAlgorithm* qui implémente l'algorithme de peintre tel que décrit dans le document de référence fourni avec l'énoncé du projet.

5. **BSPTree**

— Implémentation générale d'un arbre BSP. Basé sur les implémentation du Prof. V. Bruyère des ABR fournis sur la page Moodle du cours de SDD2. Permet de stocker des données et les sous-arbres de gauche et droite.

6. **SegmentBSPTree**

— Arbre BSP ne permettant de stocker que des segments dans les noeuds de l'arbre. Il s'agit donc de l'objet utilisé dans le projet pour représenter les arbres BSP des scènes de segments.

7. **Pair**

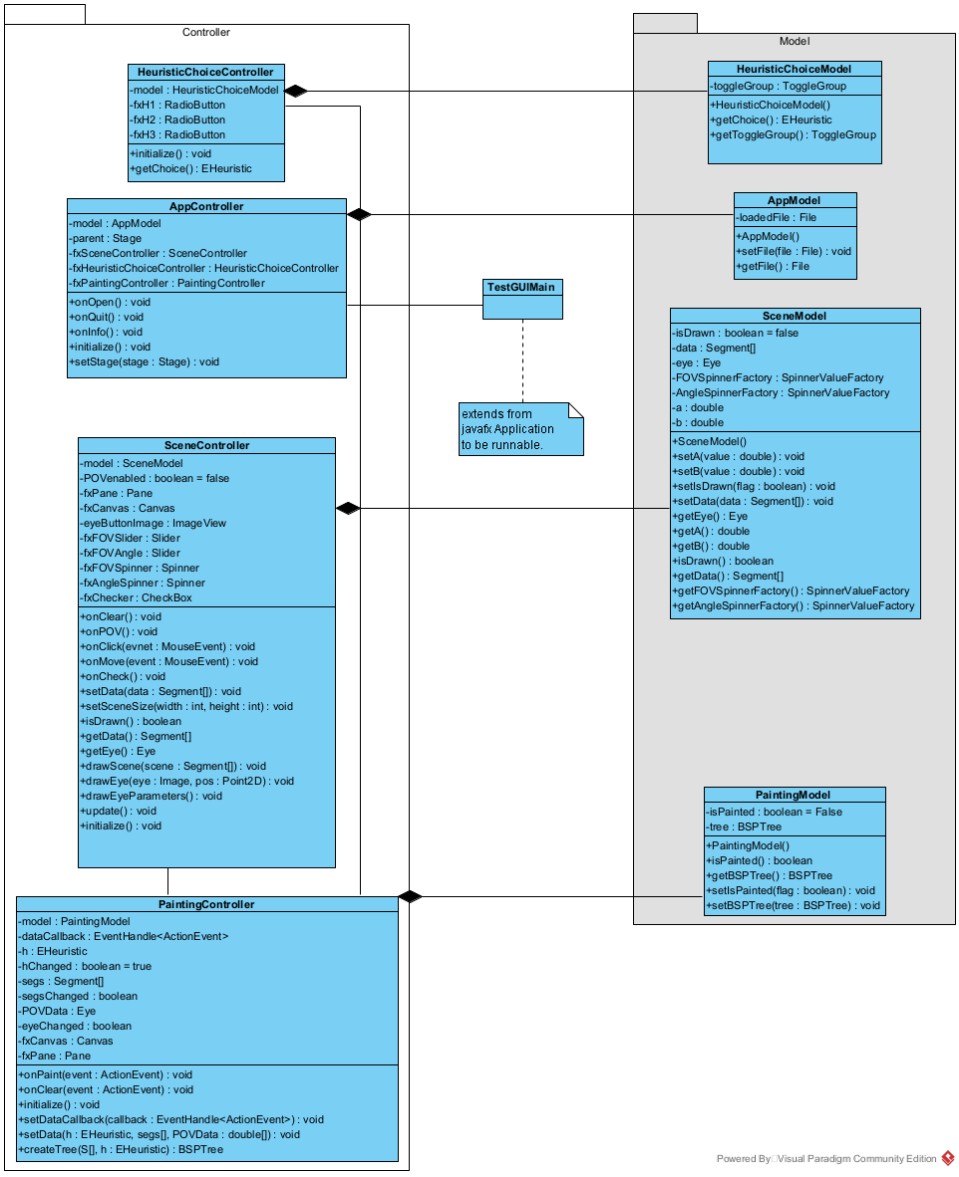
— Permet de stocker une paire de données. Très pratique pour stocker, par exemple, des coordonnées ou les deux segments créés lorsqu'un segment est coupé par un autre.

8. **Point2D**

— Comme nous venons de le dire ci-dessus, un objet Pair est utile pour stocker des coordonnées en 2D ; un objet Point2D permet donc de représenter plus spécifiquement des coordonnées.

9. **Eye**

— Représentation de l'oeil qu'on peut placer dans la scène. On retrouve alors les coordonnées, la direction vers laquelle il est tourné, la FOV, ...



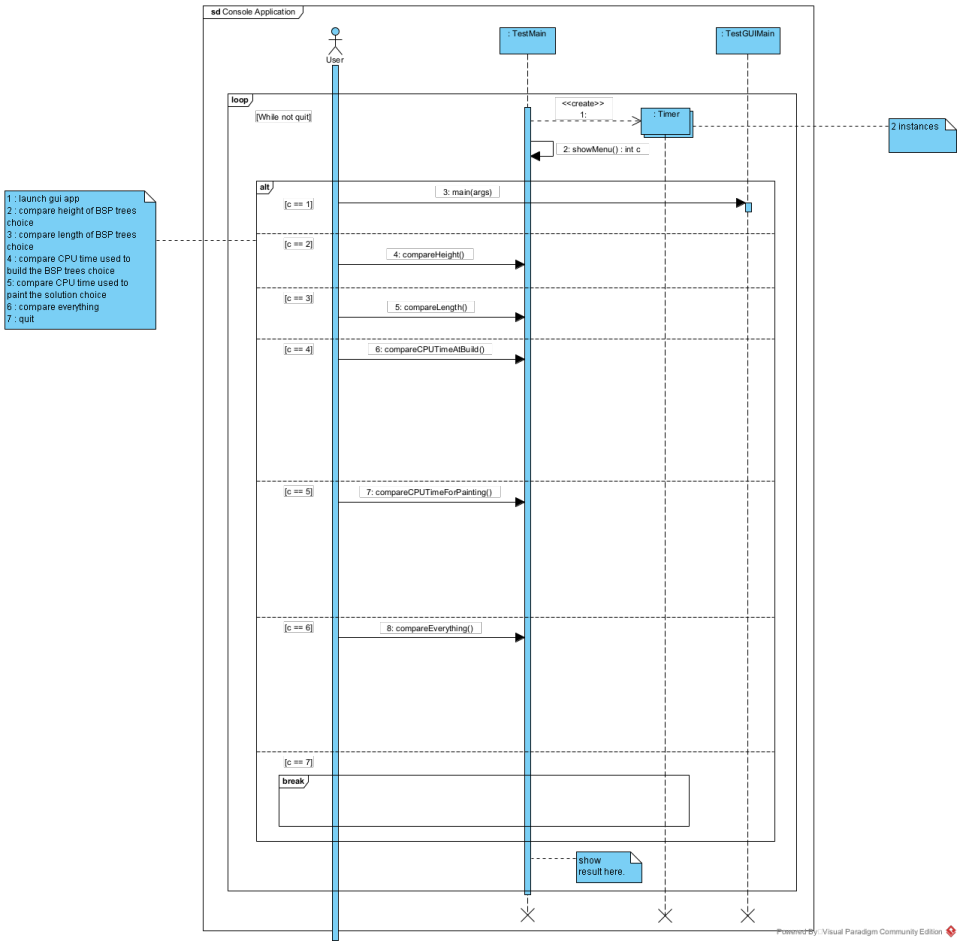

1. **TestGUIMain**
   — Implémentation principale de l'application graphique. Elle peut être lancée indépendamment de l'application console moyennant la modification d'une entrée du fichier *buil.gradle*.
2. **AppController**
   — Controlleur principale de l'application.

3. **AppModel**
   — Modèle de l'application.
4. **SceneController**
   — Contrôleur de la scène. Il s'agit du contrôleur le plus conséquent, étant celui avec lequel l'utilisateur aura le plus d'interactions (par exemple lors du placement de l'oeil).
5. **SceneModel**
   — Modèle de la scène, contenant toute les données qui peuvent être utilisées tel que l'ensemble des segments constituant ladite scène.
6. **HeuristicChoiceController**
   — Contrôleur du panneau de choix de l'heuristique.
7. **HeuristicChoiceModel**
   — Modèle du panneau de choix de l'heuristique. Contient le choix de l'utilisateur.
8. **PaintingController**
   — Contrôleur de l'endroit où sera peint la solution de ce que voit l'oeil. Partie conséquente de l'application GUI car c'est ici que sont calculés les arbres BSP, que l'algorithme du peintre est lancé, ainsi que sont calculées les projections.
9. **PaintingModel**
   — Modèle du panneau d'affichage de la solution.

## 2.3 Sequence Diagrams

Pour nous aider dans notre implémentation, nous avons établi des diagrammes de séquence pour certaines étapes pour lesquelles nous avions besoin de bien définir l'ensemble des interactions que nous allions devoir développer. Cela nous a permis avant tout de structurer plus précisément notre implémentation.

1. **Console Application**



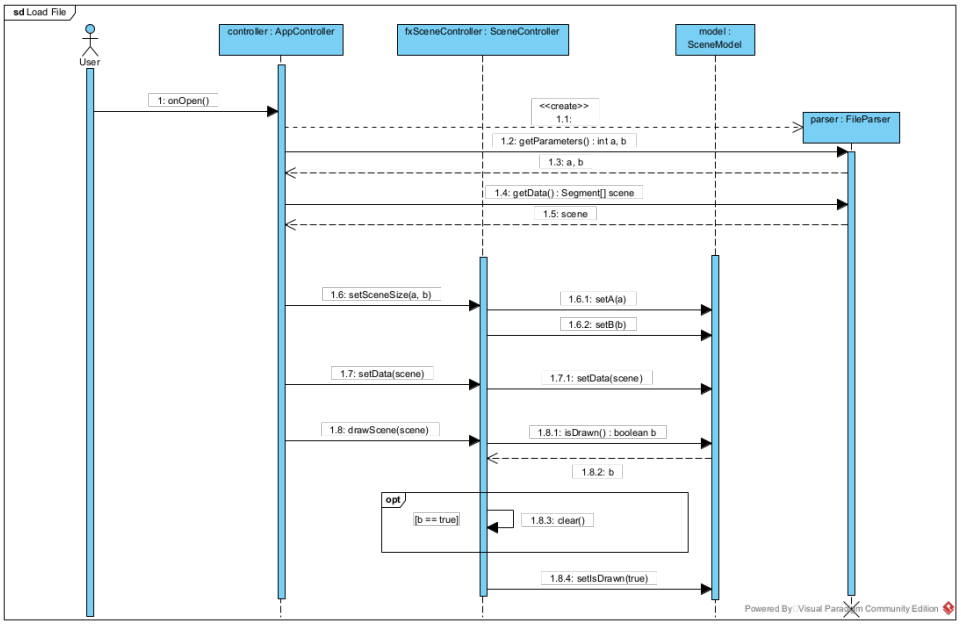

Ce diagramme représente succinctement le fonctionnement de l'application console. On y retrouve la boucle principale dans laquelle l'utilisateur peut choisir la comparaison qu'il souhaite effectuer. Nous n'avons

pas développé l'entièreté du déroulement des interactions, l'objectif étant simplement d'avoir une idée concrète de la structure que prendrait l'application. A cet effet la gestion des arguments n'y est pas décrite ni les différents choix possibles que l'utilisateur peut effectuer et leur gestion.

(Il n'a pas été mis à jour au fur et à mesure du développement de l'application et a donc été défini avant le début de la phase d'implémentation afin de donner une idée concise mais claire de la structure de cette implémentation)

Il s'initialise dès que l'application est lancée. Basiquement, l'application affiche un menu des différents choix que l'utilisateur peut faire. L'utilisateur entre l'index (explicitement écrit dans le menu) correspondant à son choix et en fonction de ce choix, le programme exécute la comparaison souhaitée : ouverture de l'application graphique, comparaison, ou quitter le programme. Si l'utilisateur choisit une comparaison, le résultat est affiché à la fin de la séquence d'actions et la boucle recommence jusqu'à ce que l'utilisateur demande l'arrêt du programme.

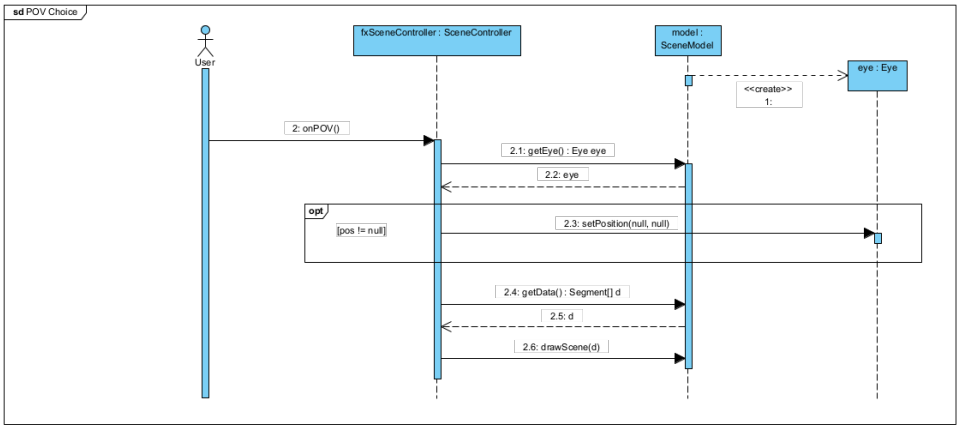
2. **Load File**




Ce diagramme décrit fidèlement les différentes interactions lors de l'ouverture d'un fichier scène, comme développé dans l'implémentation.

Il s'initialise lorsque l'utilisateur demande l'ouverture d'un fichier depuis le menu de l'application graphique. Le système propose alors à l'utilisateur un navigateur de fichier qui lui permet de choisir librement la scène à ouvrir. S'ensuit alors la lecture du fichier ligne par ligne et la création de la scène. L'application peut finalement l'afficher.

3. **POV Choice**



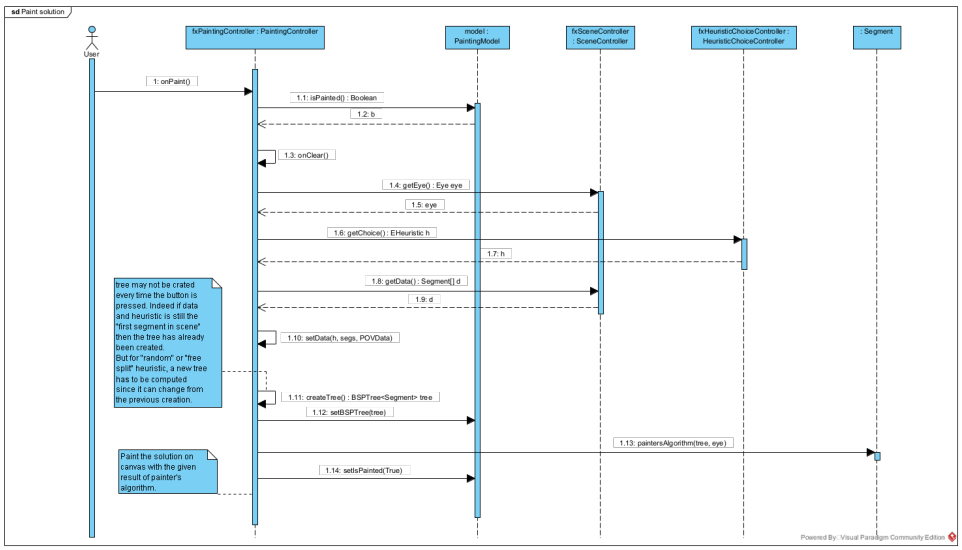



Ce diagramme permet de donner une idée du fonctionnement du placement de l'oeil sur la scène.

Il s'initialise lorsque l'utilisateur appuie sur le bouton lui permettant de placer l'oeil. l'utilisateur peut alors librement placer l'oeil sur la scène grâce à sa souris. S'il choisit d'appuyer à nouveau sur le bouton du placement de l'oeil, l'oeil n'est pas placé et l'utilisateur n'a plus la possibilité de placer l'oeil ; il faut pour cela recommencer. Sinon, il peut effectuer un clic gauche à l'endroit souhaité. Ainsi l'oeil est placé respectivement soit à la position *null* ou celle choisie. On peut alors mettre à jour la scène en y affichant l'oeil en plus des segments déjà présents.

(L'implémentation a été légèrement plus complexe étant donné qu'il fallait prendre en compte la visualisation du déplacement de l'oeil avec la souris tant qu'il n'est pas placé, ce qui est difficilement modélisable)

4. **Paint solution**




Ce dernier diagramme décrit les interactions se déroulant lorsque l'utilisateur souhaite afficher la solution, après avoir récupéré les informations nécessaires au bon fonctionnement des algorithmes.

La partie de récupération des données se passe en réalité dans un handler qui est défini dans le contrôleur de l'application (*AppController*) et ne se trouve donc pas explicitement écrit au même endroit dans l'implémentation que le reste des appels décrits dans ce diagramme.

Il s'initialise comme expliqué ci-dessus lorsque l'utilisateur appuie sur le bouton demandant l'affichage de la solution. Si une solution est déjà présente, elle est effacée. L'application récupère l'ensemble des informations nécessaires : l'ensemble des segments constituant la scène, les paramètres de l'oeil et le choix de l'heuristique. L'application peut alors créer l'arbre BSP correspondant aux données et calculer la solution grâce à l'algorithme du peintre. Finalement, la solution est affichée dans le canvas prévu à cet effet.

# 3 Algorithmes utilisés

Après avoir analysé les éléments importants constituant le projet, voici les algorithmes principaux.

**Algorithm 1** height()

**Require:** Un arbre BSP de hauteur $h$

1: **if** $isEmpty()$ **then return** 0
2: **else**
3: **return** $1 + \max(left.height(), right.height())$
4: **end if**

**Algorithm 2** size()

**Require:** Un arbre BSP de hauteur $h$

1: **if** $isEmpty()$ **then return** 0
2: **else**
3: **return** $left.length() + data.size() + right.length()$
4: **end if**

Les 2 algorithmes ci-dessus permettent respectivement de récupérer la hauteur et la taille (nombre de données) dans un arbre BSP. Il est important de noter que contrairement aux algorithmes suivants liés aux arbres BSP, ces deux-ci doivent être vus comme des méthodes de la classe permettant de représenter ces arbres.
La **première ligne** vérifie si l'arbre est vide, auquel cas la valeur de retour est 0, sinon la hauteur (respectivement la taille) de l'arbre est calculée et retournée.

**Algorithm 3** Lecture de fichier scène

**Require:** $f$ un fichier scène.

1: **if** $f \neq null$ **then**
2: $fl \leftarrow$ first file line
3: $a \leftarrow$ second word in $fl$ ▷ scene width/2
4: $b \leftarrow$ third word in $fl$ ▷ scene height/2
5: $n \leftarrow$ last word in $fl$ ▷ number of segments in scene
6: **while** there is a next line in file **do**
7: Convert lines' data into segment ▷ get segment coordinates of the two points and the color
8: **end while**
9: **end if**

Cet algorithme décrit la lecture d'un fichier scène mais nous ne tenons pas compte ici de la vérification du fichier en lui-même.
La **première ligne** vérifie que l'ouverture se soit correctement passée. Les **lignes 2 à 5** représentent la récupération des données principales de la scène (taille : largeur, hauteur ; nombre de segments) qui se trouvent dans la première ligne du fichier. Finalement, la **boucle while**, quant à elle, représente le parcours du reste du fichier afin de convertir les données en segments. La conversion se déroule de la même façon que lors de la lecture de la première ligne du fichier (récupération de la ligne, séparation de chaque donnée, assignation de ces données à une variable et ici, création d'un objet Segment à partir de ces données).

**Algorithm 4** makeTree($S$, *parent*, $freeSplit$)

**Require:** $S$, un tableau de segment ; *parent*, la référence vers le noeud (racine) parent ; $freeSplit$, un flag qui permet de dire s'il faut utiliser l'heuristique du FreeFplit ou pas.

1: $tree \leftarrow$ nouvel arbre BSP vide
2: **if** $S.length = 0$ **then return** *null*
3: **else if** $S.length = 1$ **then**
4: $tree.addData(S[0])$ ▷ ajout de la donnée dans la racine de cet arbre/sous-arbre
5: $tree.setParent(parent)$ ▷ ajout de la référence vers le noeud parent de ce sous-arbre
6: **else**
7: **if** $freeSplit$ **then**
8: $i \leftarrow 0$
9: $j \leftarrow S.length - 1$
10: *temp* ▷ variable temporaire permettant de stocker un segment pour en échanger 2 dans le set de segments $S$
11: **while** $i < j$ **do**
12: **if** $S_i$ *est un "*$freesplit$*"* **then**
13: $i \leftarrow i + 1$
14: **else**
15: $temp \leftarrow S_i$
16: $S_i \leftarrow S_j$
17: $S_j \leftarrow temp$
18: $j \leftarrow j - 1$
19: **end if**
20: **end while**
21: **if** $S_0$ $n'est\ pas\ un$ *"*$freesplit$*"* **then**
22: On mélange tout le set de segments $S$
23: **end if**
24: **end if**
25: $l \leftarrow S_0$
26: $Sminus \leftarrow Nouvelle\ liste\ chaine$
27: $Splus \leftarrow Nouvelle\ liste\ chaine$
28: $tree.addData(l)$ ▷ ajout du segment dans la racine
29: $tree.setParent(parent)$
30: **for** chaque segment *seg* dans le set $S$ **do**
31: **if** $seg! = l$ **then**
32: Ajout de *seg* dans le sous-arbre correspondant ($S^-$ ou $S^+$) en

fonction de sa position par rapport à $l$
33: **end if**
34: **end for**
35: $left \leftarrow makeTree(S^{-}.toArray(), tree, freeSplit)$
36: $right \leftarrow makeTree(S^{+}.toArray(), tree, freeSplit)$
37: $tree.setLeft(left)$
38: $tree.setRight(right)$
39: **end if**
40: **return** *tree*

Nous décrivons ici la création d'un arbre BSP. Il faut absolument un set de données $S$ représentant l'ensemble des segments qui doivent être stockés dans l'arbre. Cependant, l'argument *parent* représente le noeud parent dans l'arbre ; Celui-ci n'existant pas pour la racine principale, lors de la création de l'arbre, il suffit alors simplement de passer la valeur *null*. Enfin, le dernier argument permet l'utilisation de l'heuristique *free split* pour la création de l'arbre (valeur booléenne).
On a donc la possibilité de créer un arbre avec 3 heuristiques :

- ***Premier segment dans la liste :*** Il suffit simplement de donner le set $S$ dans l'ordre souhaité et de passer *false* comme valeur pour *free split*.
- ***Segments aléatoires :*** Il suffit simplement de mélanger l'ensemble des segments avant de le donner à cet algorithme.
- ***FreeSplit :*** Il suffit de donner la valeur *true* à l'argument *free split*.

Pour en revenir au pseudo-code, les **lignes 2 à 5** décrivent les cas de bases, quand le set $S$ est vide ou s'il n'y a qu'un seul segment dans le tableau.
De la **ligne 7 à 24**, l'algorithme est orienté uniquement sur la réalisation de l'heuristique *freSsplit*. En effet, d'abord il faut vérifier s'il existe des segments free split dans $S$, auquel cas ces segments sont ajoutés au début. Finalement, si on n'avait trouvé aucun segment *freeSplit* (si le premier segment n'est pas un *freeSplit*), il faut mélanger le jeu de données pour utiliser l'heuristique aléatoire.
A partir de la **ligne 25**, on vérifie chaque segment pour savoir s'il doit se trouver dans le sous-arbre droit ou gauche ou, avec le premier segment du set (qui permet de référence de découpe de l'espace), dans la racine de l'arbre/sous-arbre en cours de création. Après avoir finalement défini où doivent aller chacun de ces segments, il faut créer les sous-arbres correspondants grâce aux appels récursifs aux **lignes 35 et 36**.

L'algorithme se termine finalement à la **ligne 40** par le retour de l'arbre entier après avoir ajouté ses sous-arbres gauches et droits aux lignes précédentes.

**Algorithm 5** paintersAlgorithm(*tree*, *eye*)

**Require:** *tree*, un arbre BSP de hauteur $h$ et *eye* l'objet oeil.

1: **if** $tree = null$ **then**
2: **return** Liste chaînée vide
3: **else if** $tree.isLeaf()$ **then**
4: $data \leftarrow$ Nouvelle liste chaînée
5: **for** chaque segment $s$ dans $tree.getData$ **do**
6: **if** $eye.isInSight(s.getFrom())$ *or* $eye.isInSight(s.getTo())$ **then**
7: $data.add(s)$
8: **end if**
9: **end for**
10: **return** *data*
11: **end if**
12: $h \leftarrow tree.getData().get(0)$ ▷ la première donnée dans la racine de *tree*
13: **if** La FOV est supérieure à 180° **then** ▷ On doit parcourir les 2 sous-arbres obligatoirement
14: $T^- \leftarrow paintersAlgorithm(tree.getLeft(), eye)$
15: $T^+ \leftarrow paintersAlgorithm(tree.getRight(), eye)$
16: **else**
17: On vérifie si les sous-arbres gauches et droits ont un intérêt à être calculés
18: **end if**
19: **if** l'oeil est à droite dans l'arbre **then** ▷ $eye \in h^+$
20: Ajout du résultat de l'algorithme sur le sous-arbre gauche de *tree* si intéressant
21: Ajout des segments dans la racine de *tree*
22: Ajout de l'algorithme de l'algorithme sur le sous-arbre de droite de *tree* si intéressant
23: **else if** l'oeil est à gauche dans l'arbre **then** ▷ $eye \in h^-$
24: Ajout du résultat de l'algorithme sur le sous-arbre de droite de *tree* si intéressant
25: Ajout des segments dans la racine de *tree*
26: Ajout de l'algorithme de l'algorithme sur le sous-arbre gauche de *tree* si intéressant

27: **else** ▷ *eye* sur la ligne de découpe

28: Ajout des résultats de l'algorithme sur les sous-arbres de gauche et de droite ▷ Pas besoin d'afficher les segments dans la racine car ils ne sont pas vraiment visibles par l'oeil

29: **end if**

30: **return** Liste chaînée des segments de la scène que l'oeil voit, dans l'ordre dans lequel ils doivent être "peints"

---

L'algorithme ci-dessus décrit le fonctionnement de l'algorithme du peintre. Les deux **premières lignes** expriment le cas de base où il n'y a plus de segment à traiter ; de la *ligne 3 à 11*, le cas où l'arbre donné en paramètre est en réalité une feuille. Entre les **lignes 13 à 15**, on regarde le cas d'une FOV de plus de 180° ; il faut donc regarder dans les 2 sous-arbres obligatoirement. Dans les **lignes suivantes**, on décrit le cas d'une FOV de moins de 180° qui est un peu plus complexe. En effet, une optimisation serait de ne pas regarder à un sous-arbre si on est persuadé que l'oeil ne regarde pas dans la direction de l'un de ses sous-espace. Ensuite, en fonction de la position de l'oeil, on vérifie dans quel ordre les segments "intéressants" doivent être affichés.
Finalement, l'algorithme renvoie une liste de segments dans l'ordre dans lequel ils doivent être affichés.

# 4 Complexités des algorithmes

Concentrons nous maintenant sur la complexité des algorithmes présentés précédemment.

Soit $n$ le nombre de segments dans la scène.

1. **Algorithmes 1 et 2 :** Calcul de la hauteur et longueur (respectivement) d'un arbre BSP

   La vérification si l'arbre est vide se fait en $O(1)$. Le calcul du maximum et de l'addition aussi. Le coût local par noeud est donc en $O(1)$.

   Mais avant d'aller plus loin, il nous faut nous concentrer sur la taille de l'arbre, le nombre de données qui y sont stockées. En effet, on ne peut pas affirmer qu'il y n'y a que $n$ noeud dans le pire des cas car il est tout à fait possible de couper un segment en deux, par la définition de l'arbre BSP, ce qui veut dire qu'il est possible d'avoir beaucoup plus que $n$ noeuds.

   En réalité, la taille d'un arbre BSP est en $O(n \log n)$.

   **Preuve** [2] **:** Soit $s_i$ un segment de la scène. Nous devons calculer le nombre de segments qui sont coupés par la ligne $l(s_i)$, la ligne de coupe définie par $s_i$. L'ordre des segments étant important pendant la création d'un arbre BSP, la proximité d'un segment $s_j$ est noté lorsque ce dernier est coupé par $l(s_i)$.

$$dist_{s_i}(s_j) = \begin{cases} \left.\begin{array}{c}\text{nombre de segments} \\ \text{croisant } l(s_i) \text{ entre } s_i \text{ et } s_j\end{array}\right\} & \text{si } l(s_i) \text{ intersecte } s_j \\ +\infty & \text{sinon} \end{cases}$$

   Pour chaque distance finie, nous savons qu'il y a au plus 2 segments à cette distance : un de chaque côté de $s_i$.

   Soient $k = dist_{s_i}(s_j)$ et $s_{j_1}$, $s_{j_2}$ , ... , $s_{j_k}$ les segments entre $s_i$ et $s_j$. Nous pouvons calculer la probabilité que $l(s_i)$ coupe $s_j$ quand $s_i$ arrive avant $s_j$ dans l'ordre des segments des segments donnés à l'algorithme de création d'arbre (que nous analyserons un peu plus tard) et surtout avant n'importe quel segment entre $s_i$ et $s_j$. En d'autres mots, $i, j, j_1, ..., j_k$ une suite d'indices, $i$ est le plus petit.

2. M. de Berg, O. Cheong, M. van Kreveld, M. Overmars. *Computational Geometry : Algorithms and Applications.* 2008, p.265

On a donc

$$Pr[l(s_i)\ cuts\ s_j] \leq \frac{1}{dist_{s_i}(s_j)+2}$$

On peut maintenant borner le nombre total de découpes attendu, généré par $s_i$ :

$$\begin{aligned} E[\text{nombre de coupes générées par } s_i] &\leq \sum_{j \neq i} \frac{1}{dist_{s_i}(s_j)+2} \\ &\leq 2\sum_{k=0}^{n-2} \frac{1}{k+2} \\ &\leq 2\log n \end{aligned}$$

Par linéarité, on peut conclure que le nombre total de coupes attendu est au plus $2n\log n$. Puisqu'il y a $n$ segments dans la scène, au départ, le nombre total de fragments attendu est borné par $n + 2n\log n$.

□

On a donc bien la complexité suivante, dans le pire des cas, pour la taille d'un arbre :

$$O(n\log n) \tag{1}$$

On peut également affirmer que la complexité de la hauteur de l'arbre est la même étant donnée qu'il y a au plus $n+2n\log n$ noeuds (1) dans l'arbre et que la complexité local par noeud est en $O(1)$. On a donc bien que :

$$(n + 2n\log n).O(1) = O(n\log n)$$

2. **Algorithme 3 :** Lecture de fichier scène

   En supposant que la récupération d'une ligne dans un fichier, l'analyse d'une ligne et la vérification que toutes les lignes n'ont pas encore été lues sont en temps constant, on peut affirmer que chacune des opérations de l'algorithme sont en $O(1)$.

Mais la boucle *while* étant effectuées $n$ fois puisqu'il y a $n+1$ lignes dans un fichier scène (la ligne des paramètres de la scène et les n segments de la scène), la complexité de l'algorithme de lecture d'un fichier scène est en $n \times O(1)$ soit en $O(n)$.

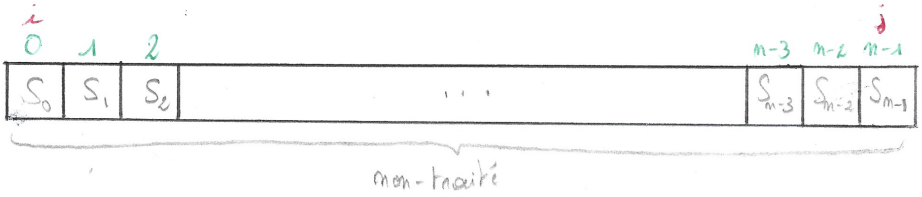
3. **Algorithme 4 :** Création d'un arbre BSP

   La création d'un arbre BSP vide ainsi que la gestion des cas de base ($S$ est vide ou ne contient qu'un seul segment), aux **5 premières lignes**, sont en temps constant.

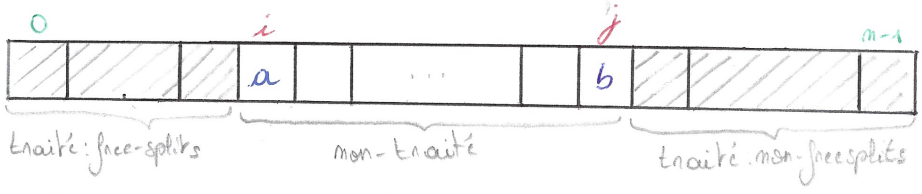
   La boucle while permet d'arranger $S$ de façon a placer un segment freesplit à la première position. Afin d'en calculer la complexité, nous allons fixer l'invariant de boucle tel que les premiers segments sont des segments freesplit s'il y en a et les autres segments sont à la fin de la liste. Dans l'algorithme que nous analysons, nous nous arrêterons, à des fins d'optimisation, si un segment freesplit est trouvé, car nous n'avons besoin que d'un seul d'entre eux étant donné que nous n'avons pas grande utilité de trier les segments suivants tant que nous sommes sûr d'avoir un segment freesplit qui serait utile à la découpe de l'espace.

   **Preuve de l'invariant :**

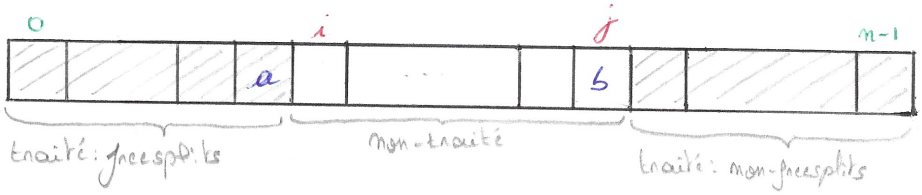
   Avant d'entrer dans la boucle, étant donné qu'aucun segment n'a encore été traité, l'invariant a été traité.



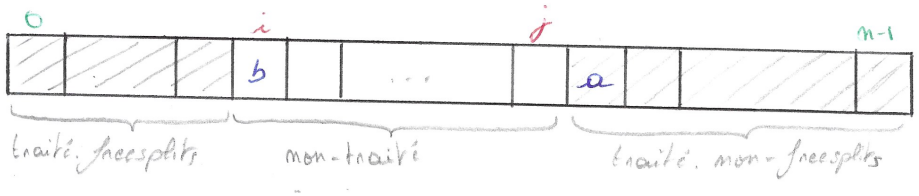

   Afin de vérifier chaque itération, supposons que l'invariant est vérifié pour la $k^{eme}$ itération. (ici, $k = i - 1$) Supposons également que les cases aux indices $i$ et $j$ on respectivement les valeurs $a$ et $b$.




   Si $a$ est un segment *freesplit*, on incrémente $i$.

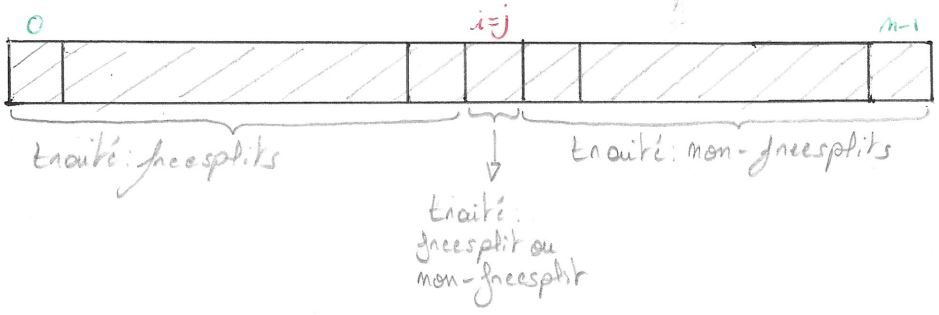



Sinon on échange les cases $i$ et $j$, afin de s'assurer que $a$, étant *non-freesplit*, se trouve dans la partie des *non-freesplits*.




On a donc bien que l'invariant est respecté, dans les 2 cas. Notons que s'il n'y avait pas de segments *freesplit* traité à la $k^{eme}$, alors $i = 0$ et l'invariant reste vérifié. C'est également vrai pour n'importe quelle itération de la boucle.

Après l'exécution complète de la boucle, nous avons bien que l'invariant est vérifié, étant donné que, s'il y en a, les segments *freesplit* se trouvent à l'avant de la liste et les segments *non-freesplit* se trouvent à l'arrière. Notons également que si aucun segment *freesplit* n'a été trouvé, $i = j = 0$.




□

L'invariant est bien vérifié et vu que l'échange de position est en temps constant et qu'on parcourt, dans le pire des cas, toute la liste, c'est-à-dire les $n$ segments, On peut alors affirmer avec certitude que la boucle while est en $O(n)$.

Le mélange de $S$ dans le cas où aucun des segments n'est un freesplit est en O(n).

On a donc, pour la condition de la **ligne 7** qu'elle est en $O(n)$.

Des **lignes 25 à 29**, les instructions sont en $O(1)$ tout comme les instructions effectuées dans la boucle $for$. Cette dernière étant parcourue autant de fois qu'il y a de segments dans $S$, on peut alors dire que la complexité de cette boucle est en $O(n)$.

Pour ce qui est des appels récursifs, nous savons que chaque noeud doit être traité. On pourrait avoir tendance à dire que l'algorithme est parcouru $n$ fois mais nous avons vu en (1) le nombre de segments maximum à traiter. On parcourt donc, dans le pire des cas, $n+2n\log n$ fois un algorithme dont le coup local est en $O(n)$. L'algorithme récursif de création d'un arbre BSP est donc en $O(n^2 \log n)$. Ce résultat semble tout à fait cohérent étant donné qu'un arbre BSP dérive d'un arbre binaire de recherche (ABR) et que nous avons vu lors des cours que la complexité de l'algorithme de tri d'un ABR, pour $n$ données, est en $O(n^2)$ ; Le coût local, pour les 2 algorithmes de tri sont en $O(n)$ mais la différence entre les 2 vient du nombre de données à traiter.

4. **Algorithme 5 :** Algorithme du peintre

   La gestion du cas d'un arbre vide est en temps constant. Mais lorsque l'arbre n'est qu'une feuille (un seul noeud), la complexité est en $O(n)$. En effet, dans le pire des cas, l'arbre BSP est constitué d'un seul noeud dans lequel tous les segments de la scène sont stockés ($n$ car il n'y a donc pas de partie coupée). Cette situation peut être représentée par une scène dans laquelle tous les segments sont compris dans la même ligne. La boucle $for$ est donc calculée $n$ fois tout au plus. Notons ce résultat (2) car nous en aurons besoin plus tard également.

   Dans le cas d'un arbre non nul ou n'étant pas réduit à une feuille, la vérification de la FOV de l'oeil est en temps constant ainsi que la vérification de l'intérêt du calcul de l'algorithme sur les sous-arbres droit et gauche. Nous nous attarderons sur les appels récursifs aux **lignes 14 et 15** un peu après ; pour l'instant nous nous concentrons sur le coût local dans l'algorithme.

   L'ajout des données en fonction de la position de l'oeil se calcule en $O(n)$ car l'ajout des données des sous-arbres se fait en temps constant car on suppose que l'appel récursif renvoie un résultat dans lequel les

segments ont déjà été triés et l'ajout de l'ensemble des données d'une liste chaînée dans une autre liste chaînée se fait en temps constant et par (2) pour la gestion des segments dans la racine.

Revenons-en maintenant aux appels récursifs. Chaque noeud de l'arbre BSP passé en paramètre étant soumis à l'algorithme, et le coût local par noeud, par les justifications ci-dessus, est en $O(n)$ dans le pire des cas, nous pouvons affirmer que la complexité totale de l'algorithme est en $n \log n.O(n) = O(n^2 \log n)$ par (1).

Nous aurions pu améliorer la complexité de cet algorithme en ne vérifiant pas, pour chaque noeud, si l'oeil voit les segments qu'il contient. En effet, la liste des données contenues dans un noeud de l'arbre étant une liste chaînée également, on aurait pu ajouter directement l'ensemble des segments à la liste des segments à retourner, ainsi la complexité de l'ajout des segments aurait été en temps constant et la complexité totale de l'algorithme en $O(n \log n)$. Mais en fonction des paramètres de l'oeil, l'augmentation de la complexité en temps permet de diminuer le nombre de données devant être stockées et données à l'algorithme qui permet d'afficher la projection de ces segments. De plus, nous estimons qu'il est plus probable de n'avoir qu'une faible quantité de segments, voir un seul, stockés dans un noeud. Finalement, il aurait de toute façon été nécessaire d'effectuer ce "scan" plus tard soit par l'algorithme d'affichage des segments, qui était déjà en $O(nlogn)$ si on suppose que l'affichage est en temps contant, puisqu'on a au plus $n + 2n \log n$ (1) segments à traiter, soit seulement appliquer la vérification pour filtrer la liste des segments, pour l'application console, ce qui nous aurait amené à créer un autre algorithme qui lui même aurait été en $O(n \log n)$ par (2) et parce qu'il y aurait également eu au plus $n+2n \log n$ (1) segments à traiter.

Ainsi nous estimons qu'en moyenne, cette "optimisation" n'en est pas une, mais ne savons pas comment le prouver de façon rigoureuse.

# 5 Comparaison des heuristiques

Pour une petite scène, le FreeSplit n'est pas une grande amélioration en comparaison à Random concernant la taille de l'arbre. En revanche, le temps CPU prit pour construire l'arbre ainsi que celui pour le peindre est plus court avec le FreeSplit.

De même, pour les grandes scènes, les tailles des arbres ne diffèrent pas significativement entre les 2 heuristiques. Cependant, l'arbre construit à l'aide du FreeSplit requiert un peu plus de temps.

On observe donc que les heuristiques Random et FreeSplit sont une bonne optimisation en comparaison au premier segment dans la scène mais, même si le FreeSplit apporte en moyenne une amélioration de la taille de l'arbre par rapport à Random, cette amélioration n'est pas significative et l'arbre prend généralement plus de temps à la construction. Mais nous observons aussi que l'arbre issu du FreeSplit permet une exécution plus rapide pour l'algorithme du peintre.

Pour appuyer nos affirmations, la page suivante contient un tableau reprenant les résultats obtenus lors de l'exécution des scènes mises à notre disposition. Ces données ont toutes été calculées avec le même ordinateur, avec les mêmes performances et dans les mêmes configurations. H1 représente l'heuristique aléatoire, H2 l'heuristique des premiers segments en premier lieu, et H3 celle du FreeSplit.

| scène | nombre de segments | heuristique | hauteur | taille | temps CPU construction arbre BSP | temps CPU algorithme du peintre |
|---|---|---|---|---|---|---|
| | | H1 | ~510 | ~4560 | ~37ms | ~20ms |
| | | H2 | 4500 | 4500 | ~30ms | ~77ms |
| ellipsesLarge | 4500 | H3 | ~515 | ~4550 | ~37ms | ~40ms |
| | | H1 | ~135 | ~750 | ~7ms | ~7ms |
| | | H2 | 720 | 720 | ~27ms | ~18ms |
| ellipsesMedium | 720 | H3 | ~140 | ~745 | ~4ms | ~5ms |
| | | H1 | ~60 | ~210 | ~2.6ms | ~2.7ms |
| | | H2 | 200 | 200 | ~5ms | ~3ms |
| ellipsesSmall | 200 | H3 | ~60 | ~210 | ~2ms | ~1.5ms |
| | | H1 | 9 | ~15 | <1ms | <1ms |
| | | H2 | 11 | 14 | 0.1ms | <1ms |
| octangle | 14 | H3 | 9 | ~16 | <1ms | <0.5ms |
| | | H1 | 8 | 8 | <1ms | <1ms |
| | | H2 | 8 | 8 | <1ms | ~0.5ms |
| octogone | 8 | H3 | 8 | 8 | <0.5ms | <0.5ms |
| | | H1 | ~50 | ~57650 | ~150ms | ~140ms |
| | | H2 | 172 | 73593 | ~300ms | ~175ms |
| randomHuge | 47017 | H3 | ~50 | ~57600 | ~220ms | ~90ms |
| | | H1 | ~45 | ~29160 | ~85ms | ~65ms |
| | | H2 | 129 | 37617 | ~140ms | ~90ms |
| randomLarge | 23504 | H3 | ~45 | ~29160 | ~160ms | ~32ms |
| | | H1 | ~40 | ~14360 | ~46ms | ~40ms |
| | | H2 | 100 | 17265 | ~90ms | ~41ms |
| randomMedium | 11045 | H3 | ~35 | ~14200 | ~40ms | ~24ms |
| | | H1 | ~30 | ~4005 | ~23ms | ~22ms |
| | | H2 | 57 | 4582 | ~35ms | ~15ms |
| randomSmall | 2939 | H3 | ~30 | ~4000 | ~15ms | ~6ms |
| | | H1 | ~15 | ~16815 | ~35ms | ~25ms |
| | | H2 | 45 | 16800 | ~57ms | ~35ms |
| rectangleHuge | 16800 | H3 | ~16 | ~16810 | ~30ms | ~18ms |
| | | H1 | ~15 | ~5970 | ~13ms | ~17ms |
| | | H2 | 44 | 5940 | ~32ms | ~15ms |
| rectangelsLarge | 5940 | H3 | ~15 | ~5980 | ~10ms | ~7ms |
| | | H1 | ~10 | 1801 | ~4.5ms | ~7ms |
| | | H2 | 20 | 1800 | ~7ms | ~6ms |
| rectangleMedium | 1800 | H3 | ~10 | 1803 | ~5ms | ~2.5ms |
| | | H1 | ~10 | ~665 | ~6ms | ~5ms |
| | | H2 | 24 | 660 | ~3ms | ~3ms |
| rectanglesSmall | 660 | H3 | ~13 | ~670 | ~3ms | ~1.5ms |

# 6 Difficultés rencontrées lors de l'implémentation

Nous avons, dès le début de l'implémentation, travaillé en *test driven development*, nous permettant d'identifier et de corriger immédiatement la majorité de nos erreurs. Malgré tout, nous avons fait face à quelques problèmes que nous n'avions pas identifiés immédiatement.
Le premier était la conversion entre le système de coordonnée cartésien et le système de coordonnée des applications graphiques. En effet, nous avions oublié que l'axe $y$ évoluait de manière différente.
Nous avons été également confronté à une erreur de type *Stack Overflow* de la jvm lors de l'affichage de la scène "ellipsesLarge" mais nous avons immédiatement étendu la mémoire allouée à la jvm.
La dernière difficulté que nous avons rencontré se passait pendant l'utilisation de l'application jar. En effet, l'utilisation des fichiers qui se trouvaient dans le fichier des resources du projet gradle est légèrement différente que l'utilisation des fichiers ressources dans un jar. Nous avons donc dû rajouter des conditions et quelques lignes de code de vérification afin de récupérer correctement les fichiers nécessaires au bon fonctionnement de l'application.

# 7 Mode d'emploi de l'application

Après extraction de l'archive, l'application peut être lancée soit en utilisant le code source, en utilisant gradle (version 6.7 utilisée lors du développement), ou bien en utilisant l'exécutable jar contenue dans l'archive.

Pour utiliser l'exécutable, il suffit d'entrer la commande *java -jar ProjetSDD2.jar* dans un terminal. Il est également possible d'y ajouter des arguments afin de lancer un mode particulier de l'application (voir explication ci-après).

Le code source étant un projet gradle, il est possible d'utiliser les commandes suivantes : *clean*, *run*, *test*, *javadoc* et *shadowJar*. Il est impératif d'utiliser cette dernière, et non *jar*, afin de créer une nouvelle version de l'exécutable du projet. Cette version sera alors située dans le dossier *build/libs/* du projet.

## 7.1 Exécutable jar

La liste des arguments utilisables est la suivante :

- ***gui*** **:** Ouvre l'application graphique.
- ***-s path*** **:** Permet de choisir au préalable un fichier scène situé au chemin *path*.
- ***–h1 id*** **:** Permet de choisir au préalable la première heuristique utilisée. Les différentes valeurs de *id* sont les suivantes :
  - *1* L'ordre des segments de la scènes est mélangé (Random),
  - *2* Les segments de la scène sont pris dans l'ordre,
  - *3* Utilisation de l'algorithme *free split*.
- ***–h2 id*** **:** Idem
- ***–compare id*** **:** Permet de choisir le type de comparaison à effectuer. Les différentes valeurs de *id* sont les suivantes :
  - *1* Compare la hauteur des deux arbres BSP créés,
  - *2* Compare la longueur (nombre de données) des deux arbres BSP créés,
  - *3* Compare le temps CPU nécessaire pour créer chacun des deux arbres BSP,
  - *4* Compare le temps CPU nécessaire pour effectuer l'algorithme du peintre sur chacun des deux arbres BSP créés.

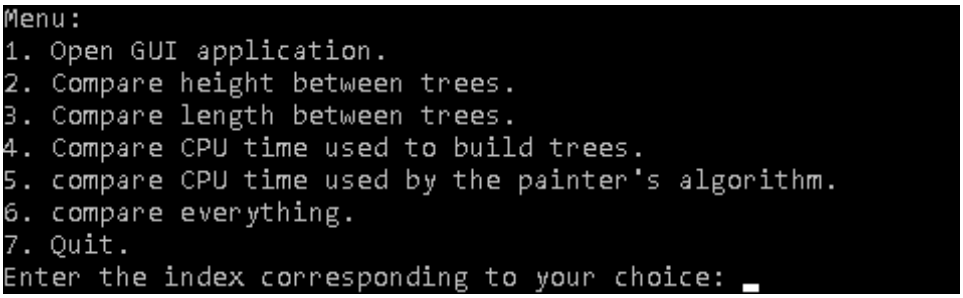
Il est important de noter que si la commande *gui* est passée en argument, aucun autre argument ne sera pris en compte.

## 7.2 Application console

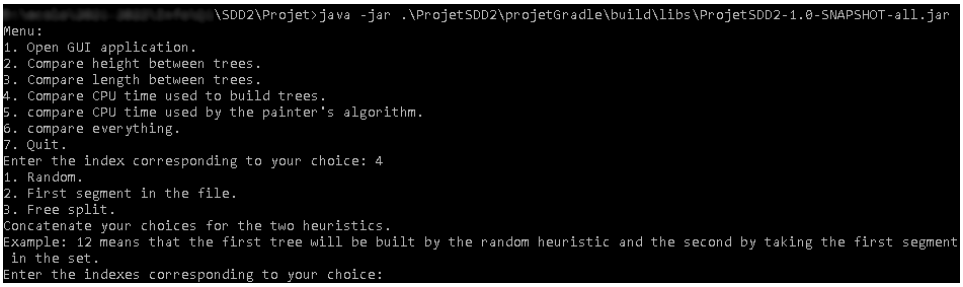
Lors de l'ouverture de l'application avec argument et s'ils sont corrects, l'application vous proposera simplement de remplir les autres champs non passés en arguments et calculera la solution.


Si aucun argument n'a été donné lors de l'exécution, L'application affiche un menu des choix disponibles de comparaison. Il suffit alors simplement de suivre les instructions.

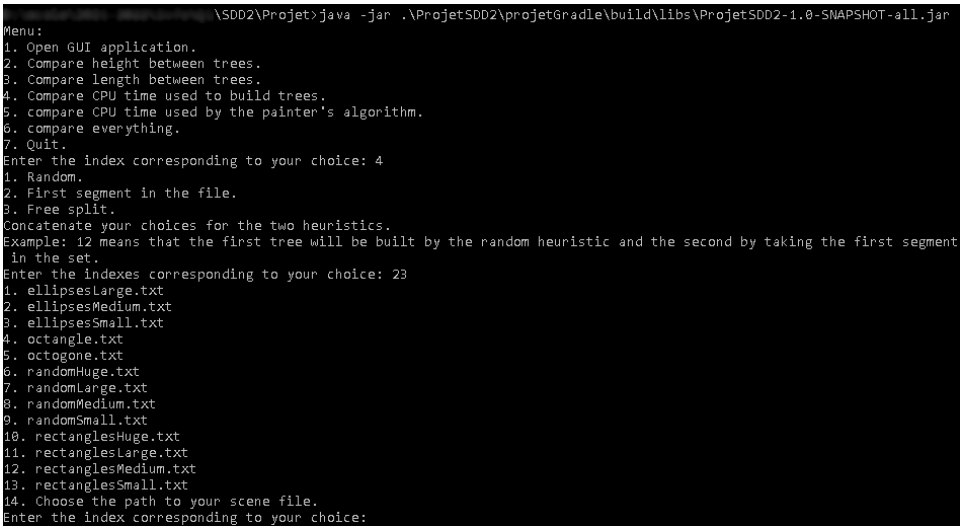
Notez qu'il est possible d'ouvrir l'application graphique à ce moment là en choisissant l'option *1*. Il est également possible de quitter l'application avec l'option *7* ou alors à tout moment lors de l'exécution du programme en utilisant la commande d'interruption *Ctrl-c*.




Après avoir choisi la comparaison qui vous convient, l'application vous demande de choisir les heuristiques que vous souhaitez utiliser pour créer vos arbres. Il est important de concaténer les indexes des heuristiques que vous souhaitez utiliser respectivement pour le choix de l'heuristique du premier et du deuxième arbre BSP à créer, comme expliqué dans l'application.




Ensuite l'application vous propose une liste de fichiers scènes prédéfinis (scènes exemples fournies pour l'implémentation et les tests de l'application) ou alors, si vous le souhaitez, de choisir un autre fichier en indiquant son chemin (*cf.* deuxième image exemple ci-après).




Finalement l'application calculera et affichera la solution comme suit :

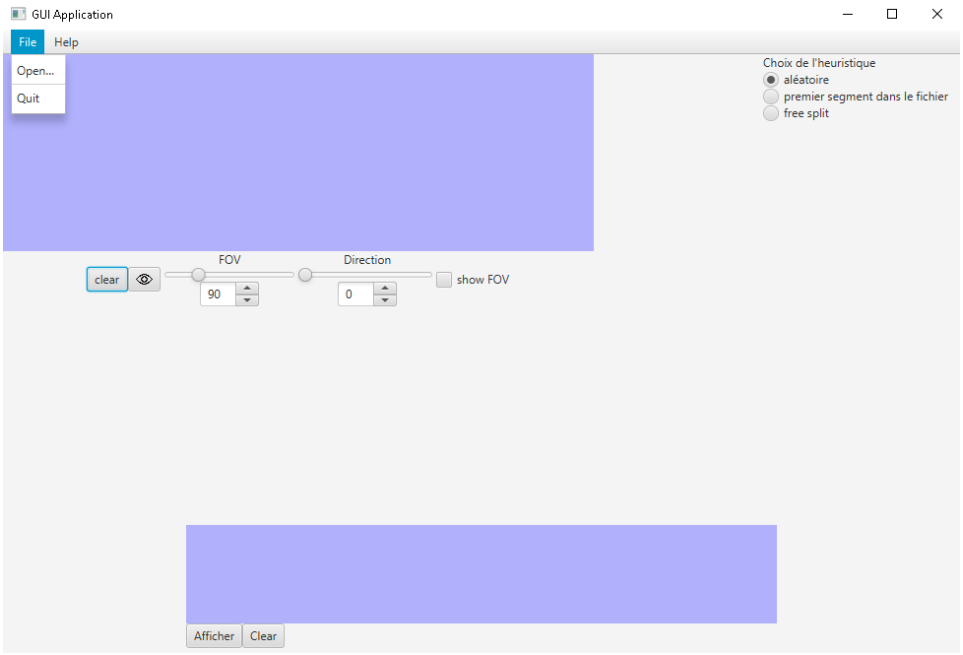
```
Enter the indexes corresponding to your choice: 23
1. ellipsesLarge.txt
2. ellipsesMedium.txt
3. ellipsesSmall.txt
4. octangle.txt
5. octogone.txt
6. randomHuge.txt
7. randomLarge.txt
8. randomMedium.txt
9. randomSmall.txt
10. rectanglesHuge.txt
11. rectanglesLarge.txt
12. rectanglesMedium.txt
13. rectanglesSmall.txt
14. Choose the path to your scene file.
Enter the index corresponding to your choice:10
building trees...
Scene: ./scenes/rectangles/rectanglesHuge.txt
width: 1250
height: 750
number of segments: 16800
Results:
        First: 6.25146E7
        Second: 2.48876E7.
All the results given for a timing comparison are given in nanoseconds.
Menu:
1. Open GUI application.
2. Compare height between trees.
3. Compare length between trees.
4. Compare CPU time used to build trees.
5. compare CPU time used by the painter's algorithm.
6. compare everything.
7. Quit.
Enter the index corresponding to your choice: _
```

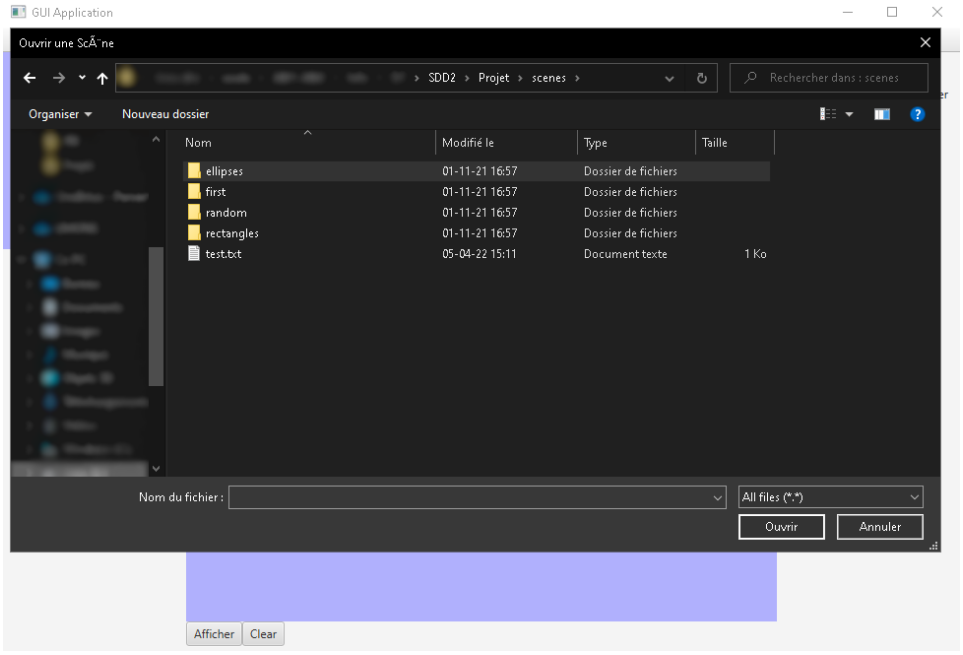
Il est donc maintenant possible de recommencer les opérations précédentes avec des valeurs différentes ou bien de quitter l'application ou encore d'essayer de visualiser la solution par vous-même en utilisant l'application graphique, grâce au menu qui est réapparu sur l'application.

## 7.3 Application graphique

A l'ouverture de l'application, tout est vide et il est uniquement possible de modifier le choix de l'heuristique ainsi que les direction et angle de vue de l'oeil qui sera placé plus tard dans la scène.

Maintenant il faut ouvrir un fichier scène à l'aide du bouton *Open* dans la catégorie *File* du menu de l'application.

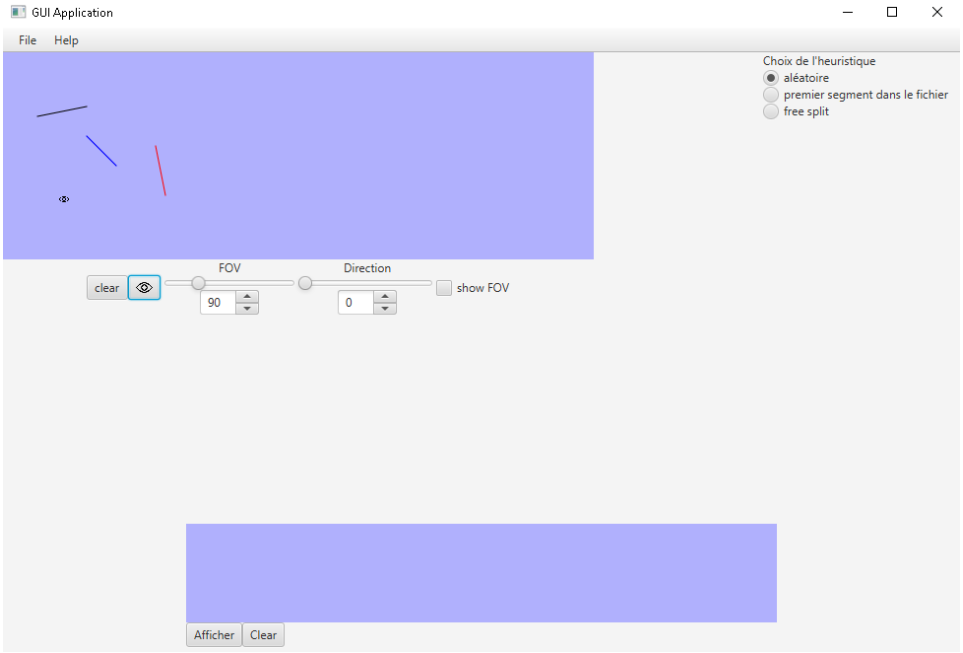


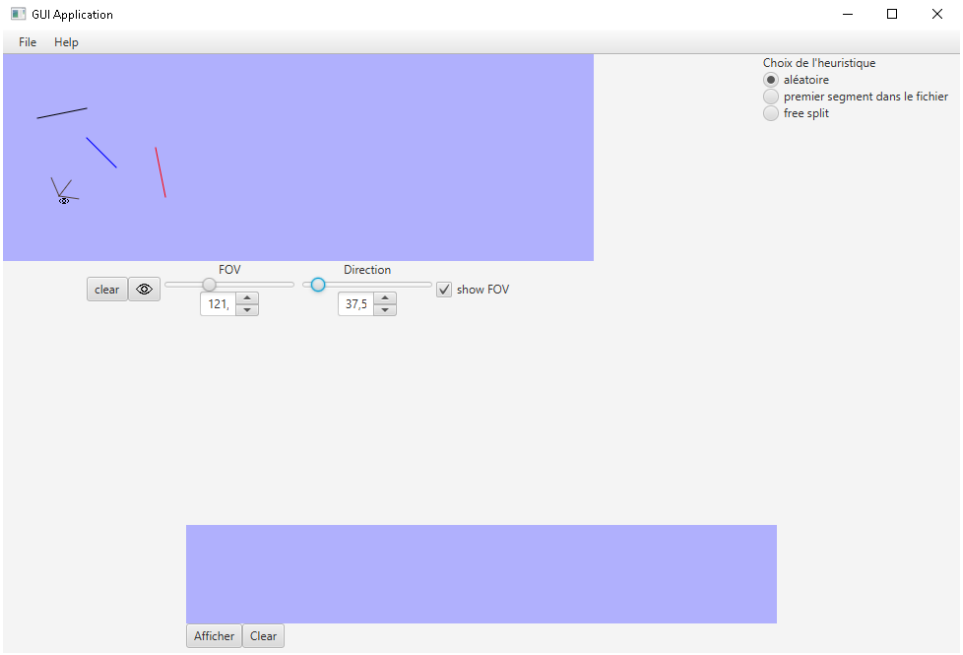

Maintenant que la scène est affichée, il ne vous reste plus qu'à y placer l'oeil. Pour ce faire, faites un clic gauche sur le bouton avec une image d'oeil,

en bas à gauche de la scène. Il vous est maintenant possible de choisir un endroit dans la scène où placer l'oeil en faisant un autre clic gauche sur la scène. Il est également possible, si vous ne souhaitez finalement pas le placer tout de suite, d'effectuer un deuxième clic gauche sur le bouton de l'oeil afin de le "ranger".

Si vous le souhaitez, vous pouvez à tout moment déplacer l'oeil en appuyant à nouveau sur le bouton de placement de l'oeil et en répétant les instructions ci-dessus.

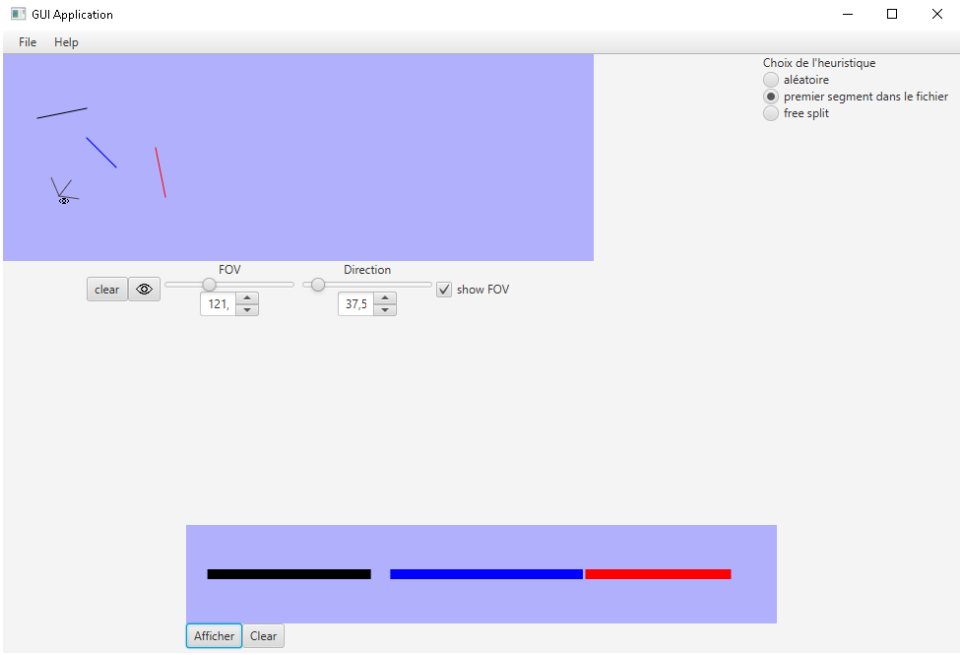



Afin de vous faciliter la visualisation des différents paramètres de l'oeil, il vous est possible de cocher la case à droite du choix de l'angle de vue de l'oeil. En faisant cela, vous afficherez ou cacherez des segments représentant la direction vers laquelle l'oeil regarde ainsi que son angle de vue.




Maintenant que vous avez choisi une scène, que vous avez placé l'oeil et paramétré sa direction et son angle de vue, et que vous avez choisi une heuristique, et seulement à ce moment-là, il vous sera possible d'afficher la projection de la solution en appuyant sur le bouton *Paint*.

Si aucune des étapes précédentes n'a été effectuée, aucune solution ne s'affichera.




Notez qu'il est possible d'effacer les affichages de la solution et de la scène grâce à leurs boutons *Clear* mais il n'est pas obligatoire de les utiliser si vous souhaitez afficher une nouvelle solution ou afficher une nouvelle scène.

En cas d'erreur de type "StackOverflow", veillez à entrez manuellement l'argument ***-Xss3m***. Il est possible, en fonction des paramètres du pc sur lequel est exécuté le fichier jar, que la mémoire de la JVM ne soit pas suffisante (comme expliqué dans la section précédente). Si par contre le code est exécuté via *gradle* et le code source, ce problème est automatiquement géré.

# Conclusions

Ce projet nous a introduit l'utilisation de structures de données plus complexes dans un contexte plus pratique et nous a permis d'appliquer des notions théoriques que nous avons apprise pendant le cours.

Rappelons également que les résultats de ce projet nous montrent que les heuristiques peuvent affecter de manière plus ou moins significative l'efficacité d'un algorithme. En l'occurrence, nous avons vu ici pour l'utilisation d'un arbre BSP que les heuristiques Random et FreeSplit sont des optimisations significatives comparées à celle constituant à prendre les segments dans l'ordre. Cependant, nous avons pu voir que les heuristiques Random et Freesplit, entre elles, n'amenaient pas à des résultats significativement différents en moyenne. Cependant, nous pouvons noter que lorsqu'on utilise l'aléatoire, l'arbre se construit plus rapidement, mais lorsqu'il s'agit d'appliquer l'algorithme du peintre, la stratégie du FreeSplit semblait généralement plus rapide.