# 12 Partitions de l'espace binaire

## L'algorithme du peintre

De nos jours, les pilotes ne font plus leur first flying expérience dans les airs, mais au sol dans un simulateur de flight. C'est moins cher pour la compagnie aérienne, plus sûr pour le pilote et meilleur pour l'environnement. Ce n'est qu'après avoir passé de nombreuses heures dans le simulateur que les pilotes sont autorisés à actionner le manche d'un véritable avion. Les simulateurs de vol doivent accomplir de nombreuses tâches différentes pour que le pilote oublie qu'il est assis dans un simulateur. Une tâche importante est la visualisation : les pilotes doivent pouvoir voir le paysage au-dessus duquel ils flissent, ou la piste sur laquelle ils atterrissent. Cela implique à la fois la modélisation des paysages et le rendu des modèles. Pour rendre une scène, nous devons déterminer pour chaque pixel de l'écran l'objet qui est visible à ce pixel ; c'est ce qu'on appelle la *suppression des surfaces cachées*. Nous devons également effectuer des calculs d'ombrage, c'est-à-dire calculer l'intensité de la lumière que l'objet visible émet dans la direction du point de vue. Cette dernière tâche prend beaucoup de temps si l'on souhaite obtenir des images très réalistes : nous devons calculer la quantité de lumière qui atteint l'objet - soit directement à partir de sources lumineuses, soit indirectement via des réflections sur d'autres objets - et prendre en compte l'interaction de la lumière avec la surface de l'objet pour voir quelle quantité est réflectée dans la direction du point de vue. Dans les simulateurs de flight, le rendu doit être effectué en temps réel, il n'y a donc pas de temps pour des calculs d'ombrage précis. Par conséquent, une technique d'ombrage rapide et simple est employée et la suppression des surfaces cachées devient un facteur important dans le temps de rendu.

L'*algorithme z-buffer* est une méthode très simple de suppression des surfaces cachées. Cette méthode fonctionne comme suit. Tout d'abord, la scène est transformée de telle sorte que la direction d'observation soit la *direction z* positive. Ensuite, les objets de la scène sont convertis par balayage dans un ordre arbitraire. La conversion par balayage d'un objet revient à déterminer les pixels qu'il couvre dans la projection ; ce sont les pixels où l'objet est potentiellement visible. L'algorithme conserve les informations relatives aux objets déjà traités dans deux tampons : un tampon d'image et un *tampon z*. Le

frame buffer stocke pour chaque pixel l'intensité de l'objet actuellement visible, c'est-à-dire l'objet qui est visible parmi ceux déjà traités. Le *z-buffer stocke* pour chaque pixel la *coordonnée z de l'objet actuellement* visible. (Plus précisément, il stocke la *coordonnée z* du point sur l'objet qui est visible à le pixel). Supposons maintenant que nous sélectionnions un pixel lors de la conversion par numérisation d'un objet.259

Si la *coordonnée z* de l'objet à ce pixel est plus petite que la *coordonnée z* stockée dans le *tampon z*, alors le nouvel objet se trouve devant l'objet actuellement visible. Nous écrivons donc l'intensité du nouvel objet dans le tampon d'image, et sa *coordonnée z* dans le *tampon z*. Si la *coordonnée z* de l'objet à ce pixel est plus grande que la *coordonnée z* stockée dans le *tampon z*, alors le nouvel objet n'est pas visible, et le tampon d'image et le *tampon z* restent inchangés. L'algorithme du *tampon z* est facilement implémenté en matériel et assez rapide en pratique. Il s'agit donc de la méthode de suppression des surfaces cachées la plus répandue. Néanmoins, l'algorithme a

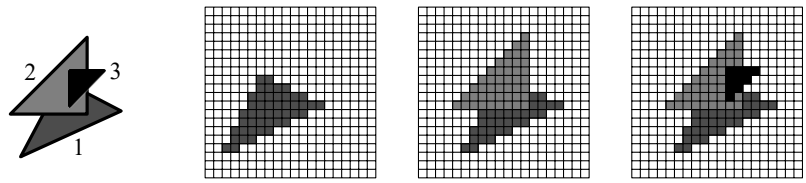
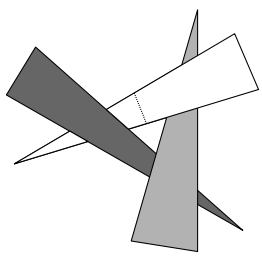



*Figure 12.1*
L'algorithme du peintre en action

quelques inconvénients : une grande quantité de stockage supplémentaire est nécessaire pour le *z-buffer*, et un test supplémentaire sur la *coordonnée z* est requis pour chaque pixel couvert par un objet. L'*algorithme du peintre* évite ces coûts supplémentaires en firmant d'abord les objets en fonction de leur distance au point de vue. Les objets sont ensuite convertis par balayage dans cet *ordre* dit *de profondeur*, en commençant par l'objet le plus éloigné du point de vue. Lorsqu'un objet est converti par balayage, nous n'avons pas besoin d'effectuer de test sur sa *coordonnée z*, nous écrivons toujours son intensité dans le tampon d'image. Les entrées du tampon d'image qui ont été filées auparavant sont simplement écrasées. La figure 12.1 illustre l'algorithme sur une scène composée de trois triangles. À gauche, les triangles sont représentés avec des numéros correspondant à l'ordre dans lequel ils sont convertis par balayage. Les images après que le first, le second et le troisième triangle ont été convertis par balayage sont également représentées. Cette approche est correcte car nous convertissons par balayage les objets dans l'ordre d'arrière en avant : pour chaque pixel, le dernier objet écrit dans l'entrée correspondante du tampon d'image sera le plus proche du point de vue, ce qui donne une vue correcte de la scène. Ce processus ressemble à la façon dont les peintres travaillent lorsqu'ils superposent des couches de peinture, d'où le nom de l'algorithme.



Pour appliquer cette méthode avec succès, nous devons être capables de trier les objets rapidement. Malheureusement, ce n'est pas si facile. Pire encore, un ordre de profondeur n'existe pas toujours : la relation in-front-of entre les objets peut contenir des cycles. Lorsqu'un tel *chevauchement cyclique* se produit, aucun ordre ne produira une vue correcte de cette scène. Dans ce cas, nous devons briser les cycles en divisant un ou plusieurs des objets, et espérer qu'un ordre de profondeur existe pour les morceaux qui résultent de la division. Lorsqu'il y a un cycle de trois triangles, par exemple, nous pouvons toujours diviser l'un d'entre eux en un morceau triangulaire et un morceau quadrilatéral, de sorte qu'un ordre d'affichage correct existe pour l'ensemble de quatre objets qui en résulte. Le calcul des objets à diviser, de l'endroit où les diviser, puis le tri des fragments d'objets est un processus coûteux. Comme

l'ordre dépend de la position du point de vue, nous devons recalculer l'ordre à chaque fois que le point de vue se déplace. Si nous voulons utiliser l'algorithme du peintre dans un environnement en temps réel tel que la simulation de flight, nous devons prétraiter la scène de manière à trouver un ordre d'affichage correct.

rapidement pour tout point de vue. Une structure de données élégante qui rend cela possible est l'arbre de partition de l'espace binaire, ou arbre BSP en abrégé.

## 12.1 La Definition des arbres BSP

Pour avoir une idée de ce qu'est un arbre BSP, regardez la figure 12.2. Cette figure montre une partition d'espace binaire (BSP) pour un ensemble d'objets dans le plan, ainsi que l'arbre qui correspond à la BSP. Comme vous pouvez le voir, la partition d'espace binaire est obtenue en divisant récursivement le plan avec une ligne : first nous divisons le plan entier avec £ 1, puis nous divisons le demi-plan au-dessus de £ 1avec £ 2et le demi-plan au-dessous de £ 1avec £ 3, et ainsi de suite. Les lignes de fractionnement ne font pas que partitionner le plan, elles peuvent aussi découper les objets en fragments. Le découpage se poursuit jusqu'à ce qu'il ne reste plus qu'un seul fragment à l'intérieur de chaque région. Ce processus est

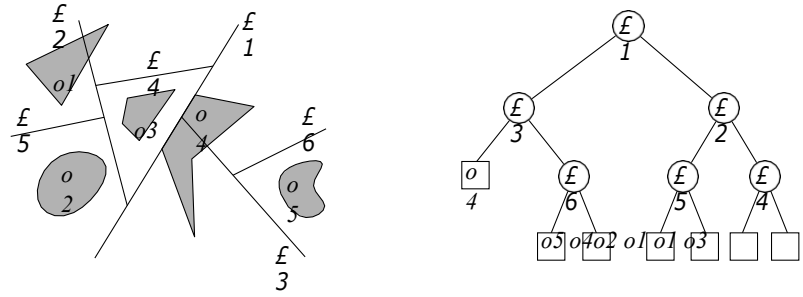



*Figure 12.2*
Une partition binaire de l'espace et l'arbre correspondant

naturellement modélisé comme un arbre binaire. Chaque feuille de cet arbre correspond à une face de la subdivision final ; le fragment d'objet qui se trouve dans la face est stocké à la feuille. Chaque nœud interne correspond à une ligne de subdivision ; cette ligne est stockée au nœud. Lorsqu'il y a des objets à 1 dimension (segments de ligne) dans la scène, alors les objets pourraient être contenus dans une ligne de fractionnement ; dans ce cas, le nœud interne correspondant stocke ces objets dans une liste.

Pour un hyperplan $h : a_1x_1 + a_2x_2 + + a_dx_d + a_{d+1} = 0$, on désigne par $h^+$le demi-espace positif ouvert délimité par $h$ et on désigne par $h^-$le demi-espace négatif ouvert :

$$h^+ := \{(x_1, x_2, \ldots, x_d) : a_1x_1 + a_2x_2 + \cdots + a_dx_d + a_{d+1} > 0\}$$

et

$$h^- := \{(x_1, x_2, \ldots, x_d) : a_1x_1 + a_2x_2 + \cdots + a_dx_d + a_{d+1} < 0\}.$$

Un arbre de partition de l'espace binaire, ou arbre BSP, pour un ensemble $S$ d'objets

dans un espace à *d dimensions* est maintenant défini comme un arbre binaire T avec les propriétés suivantes :

- Si card(*S*) ≥ 1, alors T est une feuille ; le fragment d'objet dans *S* (s'il existe) est stocké explicitement dans cette feuille. Si la feuille est désignée par ν, alors l'élément (éventuellement vide)
  L'ensemble stocké dans la feuille est noté *S*(ν)

- Si card($S$) > 1, alors la racine ν de T stocke un hyperplan $h_v$ , ainsi que l'ensemble $S$(ν) des objets qui sont entièrement contenus dans $h_v$. L'enfant gauche de ν est la racine d'un arbre BSP T $^-$pour l'ensemble $S^- := \{h_v \cap s : s \in S\}$, et l'enfant droit.
  de ν est la racine d'un arbre BSP T$^+$ pour l'ensemble $S^+ := \{h^+ \cap s : s \in S\}$.

La *taille* d'un arbre BSP est la taille totale des ensembles $S$(ν) sur tous les nœuds ν de l'arbre BSP. En d'autres termes, la taille d'un arbre BSP est le nombre total de fragments d'objets qui sont générés. Si le BSP ne contient pas de lignes de séparation inutiles - lignes qui séparent un sous-espace vide - alors le nombre de nœuds de l'arbre est au plus linéaire dans la taille de l'arbre BSP. Strictement parlant, la taille de l'arbre BSP ne dit rien sur la quantité de stockage nécessaire pour le stocker, car elle ne dit rien sur la quantité de stockage nécessaire pour un seul fragment. Néanmoins, la taille d'un arbre BSP tel que nous l'avons défini est une bonne mesure pour comparer la qualité de différents arbres BSP pour un ensemble d'objets donné.

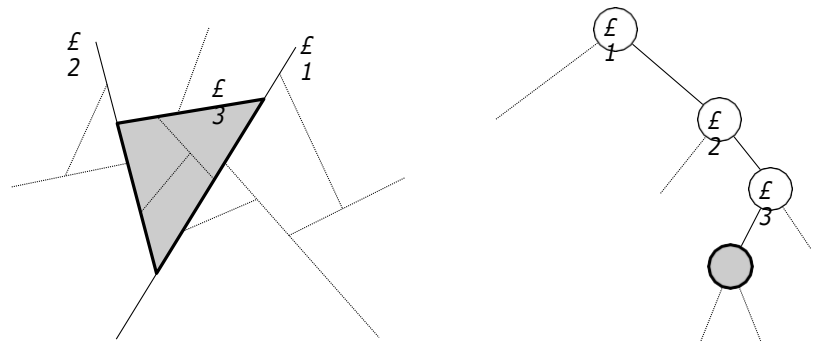
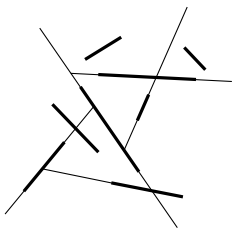



*Figure 12.3*
La correspondance entre les nœuds et les régions

Les feuilles d'un arbre BSP représentent les faces de la subdivision que le BSP induit. Plus généralement, nous pouvons identifier une région convexe à chaque nœud ν d'un arbre BSP.
Arbre BSP T : cette région est l'intersection des demi-espaces $h^{\diamond}_{\mu}$ , où $\mu$ est un ancêtre de ν et ≠quand ν est dans le sous-arbre gauche de $\mu$, et = + quand il est dans le sous-arbre gauche de μ.
est dans le sous-arbre de droite. La région correspondant à la racine de T est l'espace entier. La figure 12.3 illustre cela : le nœud gris correspond à la région grise
$£_1^+ \cap £_2^+ \cap £_3^-$.

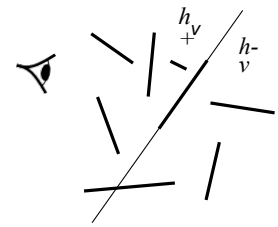


Les hyperplans de séparation utilisés dans une BSP peuvent être arbitraires. Cependant, à des fins de calcul, il peut être utile de restreindre l'ensemble des hyperplans de séparation autorisés. Une restriction habituelle est la suivante. Supposons que nous voulions construire une BSP pour un ensemble de segments de droite dans le plan. Un ensemble évident de candidats pour les lignes de séparation est l'ensemble des extensions des segments d'entrée. Un

BSP qui n'utilise que de telles lignes de séparation est appelé une *auto-partition*. Pour un ensemble de polygones planaires dans l'espace 3, une auto-partition est une BSP qui n'utilise que des plans passant par les polygones d'entrée comme plans de séparation. Il semble que la restriction aux auto-partitions soit sévère. Mais, bien que les auto-partitions ne puissent pas toujours produire des BSP de taille minimale, il est possible d'obtenir des résultats satisfaisants.
262BSP, nous verrons qu'ils peuvent en produire de raisonnablement petits.

## 12.2 Les arbres BSP et l'algorithme du peintre

Supposons que nous ayons construit un arbre BSP T sur un ensemble $S$ d'objets dans un espace tridimensionnel. Comment pouvons-nous utiliser T pour obtenir l'ordre de profondeur dont nous avons besoin pour afficher l'ensemble $S$ avec l'algorithme du peintre ? Soit $p_{\text{view}}$ le point de vue et supposons que $p_{\text{view}}$ se trouve au-dessus du plan de séparation stocké à la racine de T. Il est alors évident qu'aucun des objets situés sous le plan de séparation ne peut masquer les objets situés au-dessus. Par conséquent, nous pouvons afficher en toute sécurité tous les objets (plus précisément, les fragments d'objets) du sous-arbre $T^-$ avant d'afficher ceux de $T^+$. L'ordre des fragments d'objets
dans les deux sous-arbres $T^+$ et $T^-$ est obtenu récursivement de la même manière. C'est
résumée dans l'algorithme suivant.




**Algorithme** PAINTERSALGORITHM(T, $p_{\text{view}}$)
1. Soit v la racine de T.
2. **si** v est une feuille
3. **puis** convertir par balayage les fragments d'objets dans $S$(v).
4. **else if** $p \in h_v^+$
5. **alors** PAINTERSALGORITHM(T-, *pview*)
6. Convertir par balayage les fragments d'objets dans $S$(v).
7. ORTHODE DE PEINTURE (T+, *pview*)
8. **sinon si** $p_{\text{view}} \in h_v^-$
9. **alors** PAINTERSALGORITHM (T+), *pview*)
10. Convertir par balayage les fragments d'objets dans $S$(v).
11. GORITHME DE PEINTURE (T-, *pview*)
12. **sinon** ( $p_{\text{view}} * \in h_*$)
13. ALGORITHME DES PEINTRES , *pview*)
(T+)
14. PAINTERSALGORITHM(T-, *pview*)v

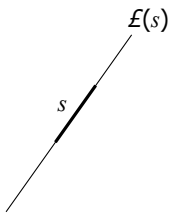

Notez que nous ne dessinons pas les polygones dans $S$(v) lorsque $p_{\text{view}}$ se trouve sur le plan de séparation $h_v$ , car les polygones sont flat objets à 2 dimensions et donc non visibles depuis les points qui se trouvent dans le plan qui les contient.

L'efficience de cet algorithme - en fait, de tout algorithme qui utilise des arbres BSP - dépend largement de la taille de l'arbre BSP. Nous devons donc choisir les plans de fractionnement de manière à ce que la fragmentation des objets soit maintenue au minimum. Avant de pouvoir développer des stratégies de fractionnement qui produisent de petits arbres BSP, nous devons décider des types d'objets que nous autorisons. Nous nous sommes intéressés aux arbres BSP parce que nous avions besoin d'un moyen rapide de supprimer les surfaces cachées pour les simulateurs flight. Comme la vitesse est notre principale préoccupation, nous devons garder le type d'objets de la scène simple : nous ne devons pas utiliser de surfaces courbes, mais tout représenter dans un

modèle polyédrique. Nous supposons que les facettes des polyèdres ont été triangulées. Nous voulons donc construire un arbre BSP de petite taille pour un
ensemble donné de triangles dans un espace tridimensionnel .263

## 12.3 Construction d'un arbre BSP

Lorsque l'on veut résoudre un problème tridimensionnel, ce n'est généralement pas une mauvaise idée de se faire une idée en firmant d'abord la version planaire du problème. C'est également ce que nous faisons dans cette section.




Soit $S$ un ensemble de $n$ segments de droite non intersectés dans le plan. Nous limiterons notre attention aux auto-partitions, c'est-à-dire que nous ne considérons que les lignes contenant un des segments de $S$ comme des lignes candidates à la division. L'algorithme récursif suivant pour construire un BSP se présente immédiatement. Soit $£(s)$ dénote la ligne qui contient un segment $s$.

**Algorithme** 2DBSP($S$)
*Entrée.* Un ensemble $S = \{s_1, s_2, \ldots, s_n\}$ de segments.
*Sortie.* Un arbre BSP pour $S$.
1. **si** card$(S) \leq 1$
2. **alors** Créer un arbre $T$ constitué d'un seul nœud feuille, où l'ensemble $S$ est stockées de manière explicite.
3. **retourner** $T$
4. **else** ( Utilisez $£(s_1)$ comme ligne de séparation. )
5. $S^+ \leftarrow \{s \cap £(s_1)^+ : s \in S\}$ ; $T^+ \leftarrow$ 2DBSP($S^+$)
6. $S^- \leftarrow \{s \cap £(s_1)^- : s \in S\}$ ; $T^- \leftarrow$ 2DBSP($S^-$)
7. Créez un arbre BSP $T$ avec un noeud racine $v$, un sous-arbre gauche $T^-$, un sous-arbre droit $T^+$, et avec $S(v) = \{s \in S : s = £(s_1)\}$.
8. **retourner** $T$

L'algorithme construit clairement un arbre BSP pour l'ensemble $S$. Mais est-ce un petit arbre ? Peut-être devrions-nous consacrer un peu plus d'efforts à choisir le bon segment pour effectuer le découpage, au lieu de prendre aveuglément le first segment, $s_1$ . Une approche qui vient à l'esprit est de prendre le segment $s \in S$ tel que $£(s)$ coupe le moins de segments possible. Mais ceci est trop gourmand : il existe des configurations de segments pour lesquelles cette approche ne fonctionne pas bien. De plus, finir ce segment prendrait beaucoup de temps. Que pouvons-nous faire d'autre ? Peut-être avez-vous déjà deviné : comme dans les chapitres précédents où nous devions faire un choix difficile, nous faisons simplement un choix aléatoire. C'est-à-dire que nous utilisons un segment aléatoire pour effectuer le fractionnement. Comme nous le verrons plus tard, le BSP résultant devrait être assez petit.

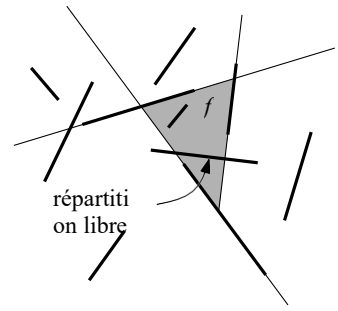
Pour ce faire, nous plaçons les segments dans un ordre aléatoire avant de commencer la construction :

**Algorithme** 2DRANDOMBSP($S$)
1. Générer une permutation aléatoire $S' = \{s_1, \ldots, s_n\}$ de l'ensemble $S$.
2. 2DBSP($S'$)
3. **retourner** $T$



Avant d'analyser cet algorithme aléatoire, nous notons qu'une opti- misation simple est possible. Supposons que nous ayons choisi les firmes lignes de partition. Ces lignes induisent une subdivision du plan dont les faces correspondent aux nœuds de l'arbre BSP que nous construisons. Considérons une telle face $f$ . Il peut

sont des segments qui traversent complètement *f*. La sélection d'un de ces segments croisés pour diviser *f* n'entraînera aucune fragmentation des autres segments à l'intérieur de *f* , tandis que le segment lui-même peut être exclu de toute considération ultérieure. Il serait stupide de ne pas profiter de ces *divisions libres*. Notre stratégie améliorée consiste donc à effectuer des divisions libres chaque fois que cela est possible, et à utiliser des divisions aléatoires dans le cas contraire. Pour mettre en œuvre cette optimisation, nous devons être en mesure de dire si un segment est une division libre. À cette fin, nous maintenons deux variables booléennes avec chaque segment, qui indiquent si les extrémités gauche et droite se trouvent sur l'une des lignes de division déjà ajoutées. Lorsque les deux variables deviennent vraies, alors le segment est une division libre.




Nous analysons maintenant la performance de l'algorithme 2DRANDOMBSP. Pour rester simple, nous analyserons la version sans splits libres. (En fait, les free splits ne font pas de différence asymptotiquement).

Nous commençons par analyser la taille de l'arbre BSP ou, en d'autres termes, le nombre de fragments qui sont générés. Bien entendu, ce nombre dépend fortement de la permutation particulière générée à la ligne 1 : certaines permutations peuvent donner de petits arbres BSP, tandis que d'autres en donnent de très grands. A titre d'exemple, considérons la permutation

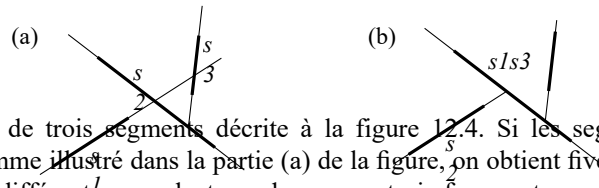


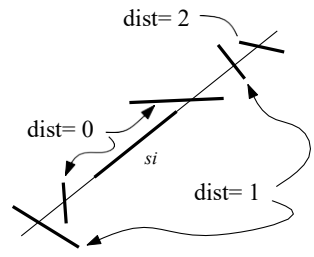

*Figure 12.4*
Des commandes différentes donnent des BSP différents

collection de trois segments décrite à la figure 12.4. Si les segments sont traités comme illustré dans la partie (a) de la figure, on obtient five fragments. Un ordre différent, cependant, ne donne que trois fragments, comme illustré dans la partie (b). Comme la taille du BSP varie en fonction de la permutation utilisée, nous allons analyser la taille *attendue* de l'arbre BSP, c'est-à-dire la taille moyenne sur l'ensemble des *n* ! permutations.

**Lemma 12.1** *Le nombre attendu de fragments générés par l'algorithme* 2DRANDOMBSP *est* $O(n \log n)$.

*Preuve.* Soit $s_i$ un segment fixé dans $S$. Nous allons analyser le nombre attendu d'autres segments qui sont coupés lorsque $\mathcal{L}(s_i)$ est ajouté par l'algorithme comme prochaine ligne de découpage.

Dans la figure 12.4, nous pouvons voir que le fait qu'un segment $s_j$ soit coupé ou non lorsque $\mathcal{L}(s_i)$ est ajouté - en supposant qu'il puisse être coupé par $\mathcal{L}(s_i)$ - dépend des segments qui sont également coupés par $\mathcal{L}(s_i)$ et qui se trouvent "entre" $s_i$ et $s_j$. En particulier, lorsque la ligne passant par un tel segment est utilisée avant $\mathcal{L}(s_i)$, elle protège $s$ de $s_j$ $s_i$. C'est ce qui s'est passé dans la figure 12.4(b) : le segment $s_a$ $_1$protégé $s_3$ de $s_2$. Ces considérations nous conduisent à définir la distance d'un segment par rapport au segment fixé $s_i$:




$$\text{dist}_{s_i}(s_j) = \begin{cases} \text{le nombre de segments qui se croisent. si } \mathcal{L}(s_i) \text{ intersecte } s_j \\ \mathcal{L}(s_i) \text{ entre } s \text{ et } s_j \\ +\infty \text{ autrement} \end{cases}$$

Pour toute distance finie, il y a au plus deux segments à cette distance, un de chaque côté de $s_i$.

Soit $k := \text{dist}_{s_i}(s_j)$, et $s_{j_1}, s_{j_2}, \ldots, s_{j_k}$ sont les segments entre $s_i$ et $s_j$. Quelle est la probabilité que $\ell(s_i)$ coupe $s_j$ lorsqu'il est ajouté comme ligne de séparation ? Pour que cela se produise, $s_i$ doit venir avant $s_j$ dans l'ordre aléatoire et, de plus, il doit venir avant n'importe quel segment entre $s_i$ et $s_j$, qui protège $s_j$ de $s_i$. En d'autres termes, de l'ensemble $\{i, j, j_1, \ldots, j_k\}$ d'indices, $i$ doit être le plus petit. Puisque l'ordre des segments est aléatoire, ceci implique

$$\Pr[\ell(s_i) \text{ coupe } s_j] \leq \frac{1}{\text{dist}_{s_i}(s_j) + 2}.$$

Remarquez qu'il peut y avoir des segments qui ne sont pas coupés par $\ell(s_i)$ mais dont le *prolongement* protège $s_j$. Ceci explique pourquoi l'expression ci-dessus n'est pas une égalité.

Nous pouvons maintenant lier le nombre total attendu de coupes générées par $s_i$:

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

Par linéarité de l'espérance, nous pouvons conclure que le nombre total attendu de coupes générées par tous les segments est au plus $2n \ln n$. Puisque nous commençons avec $n$ segments, le nombre total attendu de fragments est limité par $n + 2n \ln n$.

Nous avons montré que la taille attendue du BSP qui est généré par 2DRANDOMBSP est $n + 2n \ln n$. Par conséquent, nous avons prouvé qu'un BSP de taille $n + 2n \ln n$ *existe* pour tout ensemble de $n$ segments. De plus, au moins la moitié de toutes les permutations mènent à un BSP de taille $n + 4n \ln n$. Nous pouvons utiliser cela pour findr un BSP de cette taille : Après avoir exécuté 2DRANDOMBSP, nous testons la taille de l'arbre, et si elle dépasse cette limite, nous recommençons simplement l'algorithme avec une nouvelle permutation aléatoire. Le nombre attendu d'essais est de deux.

Nous avons analysé la taille du BSP produit par 2DRANDOMBSP. Qu'en est-il du temps d'exécution ? Encore une fois, cela dépend de la permutation aléatoire utilisée, nous examinons donc le temps d'exécution prévu. Le calcul de la permutation aléatoire prend un temps linéaire. Si nous ignorons le temps des appels récursifs, alors le temps pris par l'algorithme 2DBSP est linéaire dans le nombre de fragments dans $S$. Ce nombre n'est jamais plus grand que $n$ - *en* fait, il devient plus petit avec chaque appel récursif. Enfin, le nombre d'appels récursifs est évidemment limité par le nombre total de fragments générés, qui est $O(n \log n)$. Par conséquent, le temps de construction total est $O(n^2 \log n)$, et nous obtenons le résultat suivant.



**Théorème 12.2** *Un BSP de taille $O(n \log n)$ peut être calculé en temps attendu $O(n^2 \log n)$.*