ÉPREUVE COMMUNE DE TIPE 2010 - Partie D

TITRE :

Affichage rapide de scènes 3D

Temps de préparation : ……2 h 15 minutes
Temps de présentation devant les examinateurs : …….10 minutes
Entretien avec les examinateurs : ……..10 minutes

GUIDE POUR LE CANDIDAT :

Le dossier ci-joint comporte au total : 14 pages
Document principal (12 pages, dont celle-ci) ; annexes : 2 pages

Travail **suggéré** au candidat :

Le candidat pourra, au choix :

- fournir un algorithme global montrant l'articulation entre les différents algorithmes présentés dans ce document et faisant apparaître les sorties de chaque algorithme qui sont réutilisées en entrée de l'algorithme suivant ;
- ou bien, après une présentation brève de l'algorithme du *z-buffer*, donner une présentation détaillée de la procédure de remplissage des polygones.

Attention : s*i le candidat préfère effectuer un autre travail sur le dossier, il lui est* ***expressément recommandé*** *d'en informer le jury avant de commencer l'exposé.*

CONSEILS GENERAUX POUR LA PREPARATION DE L'EPREUVE :

* Lisez le dossier en entier dans un temps raisonnable.
* Réservez du temps pour préparer l'exposé devant les examinateurs.

- Vous pouvez écrire sur le présent dossier, le surligner, le découper … mais tout sera à remettre aux examinateurs en fin d'oral.
- En fin de préparation, rassemblez et ordonnez soigneusement TOUS les documents (transparents, *etc.*) dont vous comptez vous servir pendant l'oral, ainsi que le dossier, les transparents et les brouillons utilisés pendant la préparation. En entrant dans la salle d'oral, vous devez être prêts à débuter votre exposé.
- A la fin de l'oral, vous devez remettre aux examinateurs le présent dossier, les transparents et les brouillons utilisés pour cette partie de l'oral, ainsi que TOUS les transparents et autres documents présentés pendant votre prestation.

Affichage rapide de scènes 3D

La synthèse d'images consiste, en toute généralité, à générer des images en utilisant des ordinateurs. Cela va des primitives graphiques de base, telles que le tracé d'une droite sur un écran d'ordinateur, au calcul d'images très complexes, telles qu'on en voit dans les effets spéciaux de films à gros budget. Ce document présente des notions et algorithmes permettant de réaliser un affichage d'une scène 3D suffisamment rapide pour être utilisé dans des applications interactives. L'affichage proposé repose sur l'algorithme du *z-buffer* qui permet d'éliminer les parties cachées d'une scène. Cet algorithme ne s'applique qu'une fois la scène 3D transformée en un ensemble de polyèdres. Les algorithmes, dits de facettisation, qui permettent cette transformation, dépendent de la façon dont les objets de la scène ont été modélisés et ne sont pas présentés dans ce document.

Sur un écran d'ordinateur, une image est discrétisée sous la forme d'un tableau bidimensionnel de petits rectangles, appelés *pixels* (***pict**ure **el**ement*). Si l'image est en noir et blanc, avec 256 niveaux de gris, chaque pixel occupe un octet de mémoire, et si l'image est en couleur, il occupe trois octets (un octet pour chaque couleur primaire : rouge, vert et bleu).

1. Modélisation d'une scène 3D

La "facettisation" d'une scène 3D permet de modéliser les objets de cette scène avec des polyèdres dont les facettes sont donc des polygones dans l'espace 3D. Chaque polygone est lui-même représenté en mémoire par la liste de ses sommets.

L'image 2D, sur le plan de l'écran, peut être conçue comme une image obtenue avec une caméra virtuelle (cf. figure 1) : le plan de l'écran (noté *E*) est le *plan-image* de cette caméra ; il est situé derrière un centre optique (noté *C*) et tout point de la scène observé par la caméra se projette sur le plan-image par projection centrale par rapport au centre optique *C*.

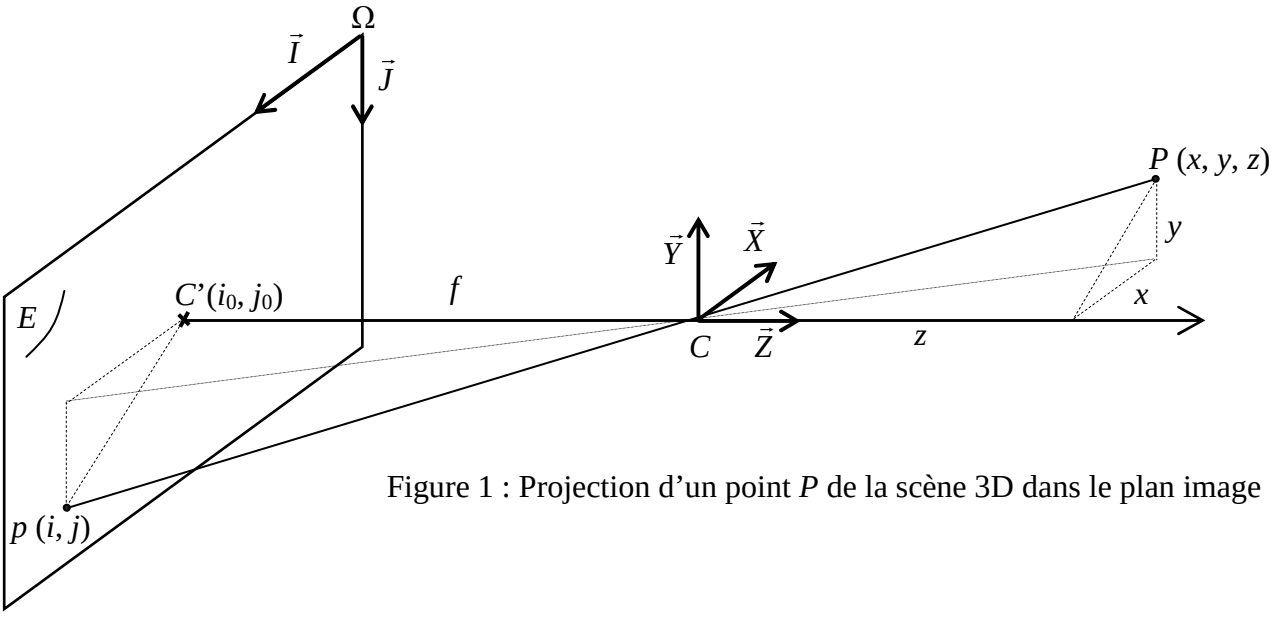


Figure 1 : Projection d'un point *P* de la scène 3D dans le plan image

La scène 3D est dotée d'un repère orthonormé direct attaché à la caméra (C, $\vec{X}$, $\vec{Y}$, $\vec{Z}$), le vecteur $\vec{Z}$ étant dirigé selon l'axe optique, orthogonal à E. De même, le plan image est doté d'un repère orthogonal (Ω, $\vec{I}$, $\vec{J}$), les vecteurs $\vec{X}$ et $\vec{I}$ étant colinéaires et de directions opposées. Ce repère est celui des coordonnées en pixels et comme les pixels ne sont pas forcément carrés, le repère n'est pas forcément orthonormal : on pose alors $\vec{I} = -r_x \vec{X}$ et $\vec{J} = -r_y \vec{Y}$, avec $r_x, r_y \in \mathbb{R}_+^*$.

En notant C' la projection orthogonale du centre optique C sur le plan image E, i_0 et j_0 les coordonnées de C' dans le repère (Ω, $\vec{I}$, $\vec{J}$) et f la distance entre les points C et C', les coordonnées i et j de la projection p sur le plan E d'un point P (x, y, z) de la scène sont données par les relations simples suivantes : $i = i_0 + \frac{f\,x}{r_x\,z}$ et $j = j_0 + \frac{f\,y}{r_y\,z}$. Ces coordonnées réelles permettent de déduire facilement sur quel pixel de l'image discrétisée sera affiché le point P.

Réciproquement, si un point p du plan image est la projection d'un point P de la scène appartenant à un polygone défini par ses sommets S_0, …, S_n, les coordonnées de P peuvent être déduites de celles de p en utilisant l'équation $Ax + By + Cz + D = 0$ du plan contenant ce polygone. En effet, avec $\overrightarrow{pC} = \overrightarrow{pC'} + \overrightarrow{C'C} = -(i - i_0)\vec{I} - (j - j_0)\vec{J} + f\,\vec{Z} = r_x (i - i_0)\vec{X} + r_y (j - j_0)\vec{Y} + f\,\vec{Z}$, on peut écrire une équation paramétrique de la demi-droite [CP) sous la forme $\overrightarrow{CP}(t) = t\,\overrightarrow{pC}$, avec $t \in \mathbb{R}_+^*$. En posant $P = P(t_0)$ et en notant que P appartient au plan d'équation $Ax + By + Cz + D = 0$, on a alors : $A\,t_0\,r_x\,(i - i_0) + B\,t_0\,r_y\,(j - j_0) + C\,t_0\,f + D = 0$. Cette relation permet de déduire la valeur de t_0, puis les coordonnées de P.

Les coefficients A, B et C de l'équation du plan contenant un polygone de sommets S_0, …, S_n ne sont autres que les coordonnées de la normale au plan et peuvent donc être calculés comme le produit vectoriel de deux vecteurs non parallèles contenus dans ce plan. Pour calculer le quatrième coefficient, D, il suffit d'écrire que le point S_0 (x_0, y_0, z_0) appartient au plan.

2. L'algorithme du z-buffer

Après la "facettisation", la scène 3D peut être considérée comme un ensemble de polygones dans l'espace. Du point de vue d'un observateur positionné sur le point C regardant dans la direction du vecteur $\vec{Z}$ certains de ces polygones peuvent être cachés par d'autres. L'algorithme du *z-buffer* permet de ne pas afficher les parties cachées. Son idée générale est la suivante : un pixel p d'indices (i, j) peut être la projection de plusieurs points P_1, …, P_n appartenant à des polygones Π_1, …, Π_n ; dans ce cas, sa couleur doit être celle du point le plus proche de C, c'est-à-dire celle du point dont la coordonnée z est la plus petite.

En considérant un écran dont la définition est de j_{max} lignes et de i_{max} colonnes, l'algorithme va utiliser un tableau bidimensionnel de même taille, dit "*z-buffer*", noté `Min`, tel que `Min` [*i*, *j*] correspondra au minimum de la profondeur *z* des points des objets de la scène rencontrés qui se projettent sur le pixel d'indices (*i*, *j*).

L'algorithme de principe est donc le suivant :

1. Initialiser la couleur des pixels à la couleur du fond ;
2. Initialiser les valeurs du *z-buffer* Min à +∞ ;
3. Pour chaque polygone Π de la scène 3D :
   3.1. Calculer la projection de tous ses sommets sur le plan image ; on obtient un polygone π dans le plan image ;
   3.2. Déterminer l'intersection de π avec la fenêtre rectangulaire *E* que constitue l'écran de l'ordinateur ;
   On obtient un nouveau polygone π' = π ∩ *E* (cf. figure 2) ;
   3.3. Pour chaque pixel *p*(*i*, *j*) à l'intérieur de π' :
      3.3.1. Calculer le point *P* de Π qui se projette sur *p* ;
      3.3.2. Si la coordonnée *z* du point *P*, notée *P*.*z*, est positive et inférieure à Min [*i*, *j*], alors Min [*i*, *j*] prend pour valeur *P*.*z* et le pixel *p*(*i*, *j*) prend la couleur du point *P*.

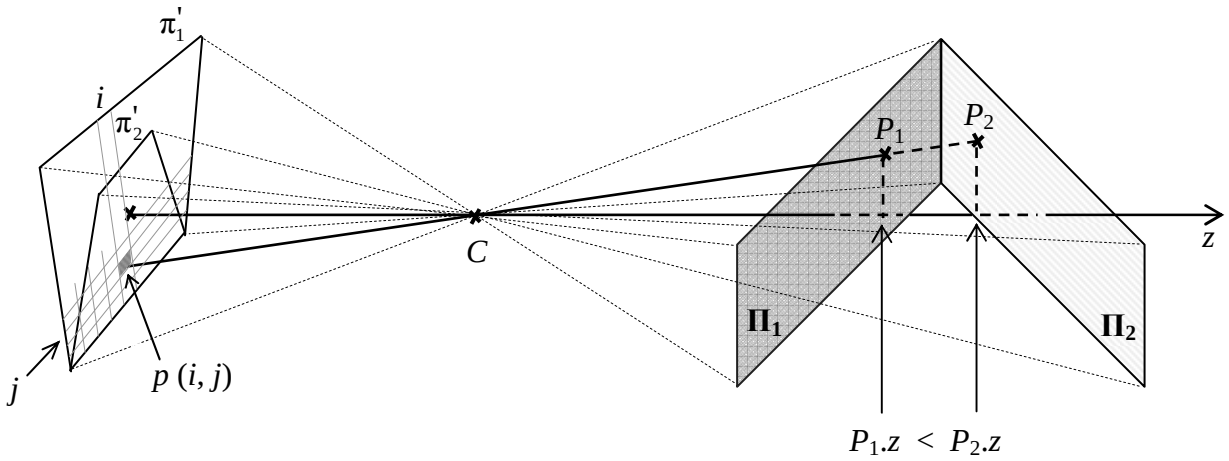


Figure 2 : Principe de l'algorithme du *z-buffer*

Le point 3.2 de l'algorithme est réalisé en appliquant un algorithme de fenêtrage (cf. section 3).

Le point 3.3 de l'algorithme est réalisé en appliquant et en adaptant un algorithme de remplissage d'un polygone (cf. section 4).

3. L'algorithme de fenêtrage

Le but d'un algorithme de fenêtrage est de déterminer, en général, l'intersection π' d'un polygone π et d'un polygone convexe *E*, représentant une "fenêtre" à travers laquelle on voit le polygone π. Pour l'algorithme du *z-buffer*, cette fenêtre est l'écran de l'ordinateur, et *E* est donc un rectangle ($E_0E_1E_2E_3$) dans le plan image (cf. figure 3).

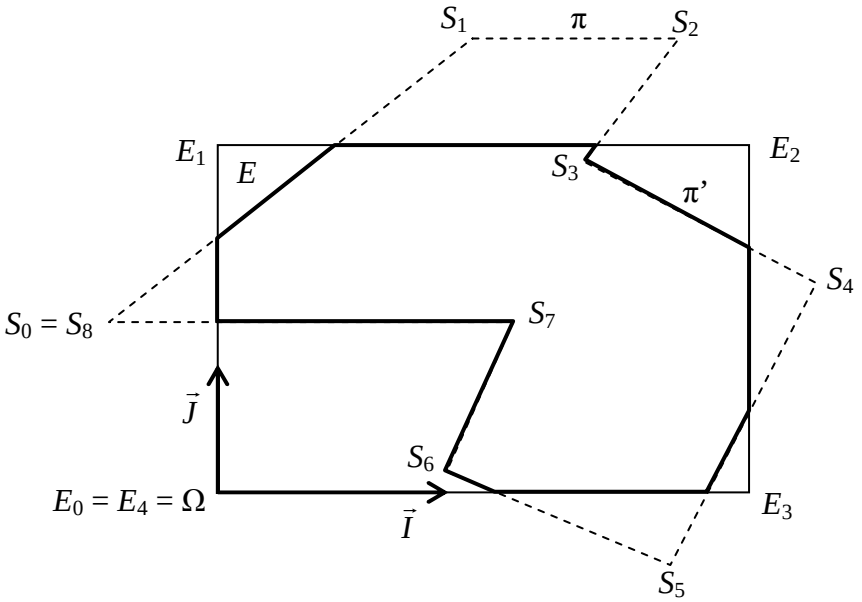


Figure 3 : Exemple de fenêtrage pour une fenêtre rectangulaire

Soit une droite (*AB*) du plan, que l'on oriente arbitrairement de *A* vers *B*. Un point *M* est dit situé *à droite* (respectivement *à gauche*) de (*AB*) si et seulement si $\det(\overrightarrow{AB}, \overrightarrow{AM}) < 0$ (respectivement $\det(\overrightarrow{AB}, \overrightarrow{AM}) > 0$). Si les sommets du polygone convexe *E* sont indicés en tournant dans le sens trigonométrique inverse (comme cela est le cas sur la figure 3), la partie du plan limitée par *E* est l'intersection des demi-plans F_i formés par les points situés à droite des droites (E_iE_{i+1}), pour $i \in [\![0, 3]\!]$ (en remarquant que $E_0 = E_4$). Il faut donc déterminer l'intersection de π avec chaque demi-plan F_i.

Un polygone π peut être représenté par la liste de sommets $(S_0, \ldots, S_{n-1}, S_n)$, en posant $S_n = S_0$. La détermination de $\pi \cap F_i$ nécessitera éventuellement le calcul de $[S_jS_{j+1}] \cap (E_iE_{i+1})$.

L'algorithme de fenêtrage de π dans une fenêtre rectangulaire *E* peut être écrit ainsi :

```
De i = 0 à 3, faire :                                  // calcul de π' = π ∩ Fi //
1. Initialiser π' à la liste vide ;
2. De j = 0 à n-1, faire :
   2.1. Si det (EiEi+1 , EiSj ) ≤ 0 alors                                  // Sj ∈ Fi //
      2.1.1. Rajouter Sj dans π' ;
      2.1.2. Si det (EiEi+1 , EiSj+1 ) > 0 alors                           // Sj+1 ∉ Fi //
         2.1.2.1. Calculer Ij = [SjSj+1] ∩ (EiEi+1) ;
         2.1.2.2. Rajouter Ij dans π' ;
             Fin si
   2.2. sinon                                                              // Sj ∉ Fi //
      2.2.1. Si det (EiEi+1 , EiSj+1 ) ≤ 0 alors                           // Sj+1 ∈ Fi //
         2.2.1.1. Calculer Ij = [SjSj+1] ∩ (EiEi+1) ;
         2.2.1.2. Rajouter Ij dans π' ;
             Fin si
3. π ← π'                                              // π prend pour valeur π' //
Retourner π'.
```

Avec l'exemple donné en figure 3, les états successifs du polygone π' sont donnés dans la figure 4 ci-dessous :

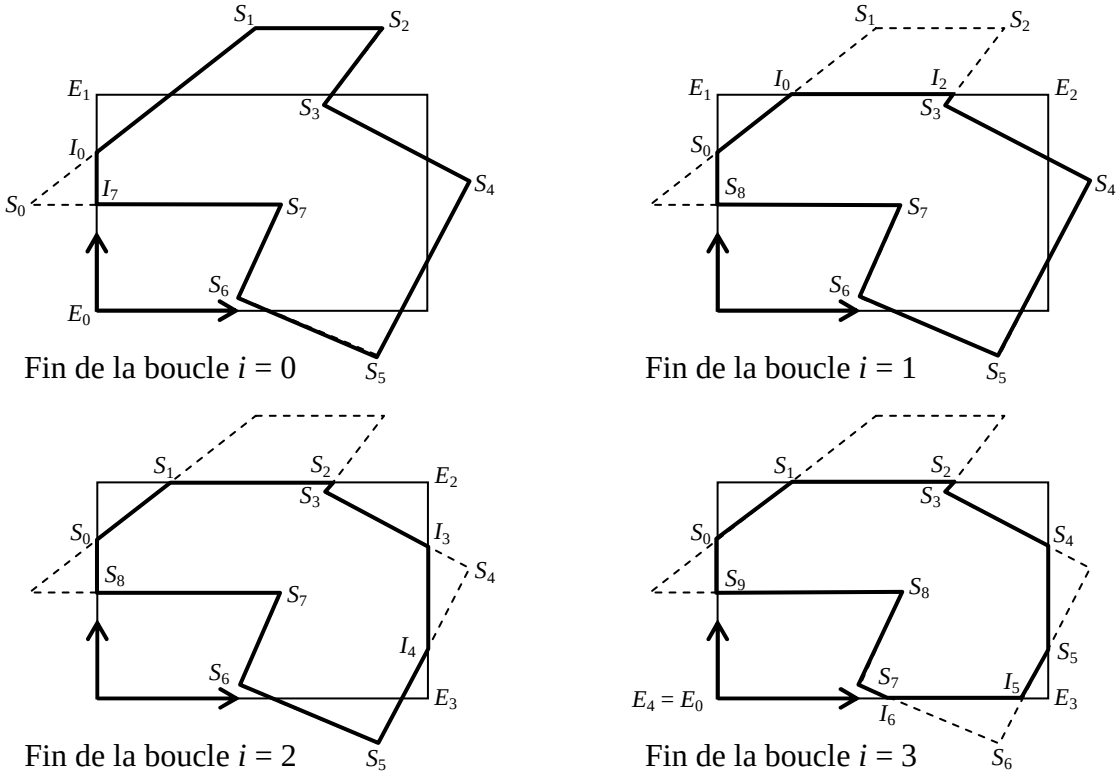


Figure 4 : Les états successifs du polygone π' au cours de l'algorithme de fenêtrage

4. L'algorithme de remplissage d'un polygone

4.1 *Principe général*

Ce type d'algorithme permet, étant donné un polygone dont les sommets ont des coordonnées entières, de parcourir tous les points de coordonnées entières du plan qui sont intérieurs à celui-ci (un point *M* est dit *intérieur* à un polygone π si et seulement si toute demi-droite issue de *M* et ne contenant aucun sommet de π intersecte un nombre impair d'arêtes de π). Une utilisation classique de cet algorithme consiste à afficher tous les points à l'intérieur du polygone.

Le principe du remplissage consiste à construire des segments horizontaux inclus dans le polygone. Par exemple, pour le polygone représenté figure 5, sur l'horizontale $y = 7$ on a deux segments horizontaux inclus dans le polygone, placés entre les abscisses 2 et 9,5 et 10,5 et 18.

L'algorithme de principe est le suivant :

Pour chaque ligne horizontale :

1. Trouver les extrémités des segments horizontaux inclus dans le polygone. Ces extrémités correspondent à des intersections de la ligne horizontale avec les arêtes du polygone.
2. Trier les extrémités obtenues à l'étape 1 dans l'ordre des abscisses croissantes : on obtient une liste de longueur paire (x_0, …, x_{2n-1}).
3. Pour k = 0 à n-1 : afficher tous les pixels d'abscisse entière comprise entre x_{2k} et x_{2k+1}.

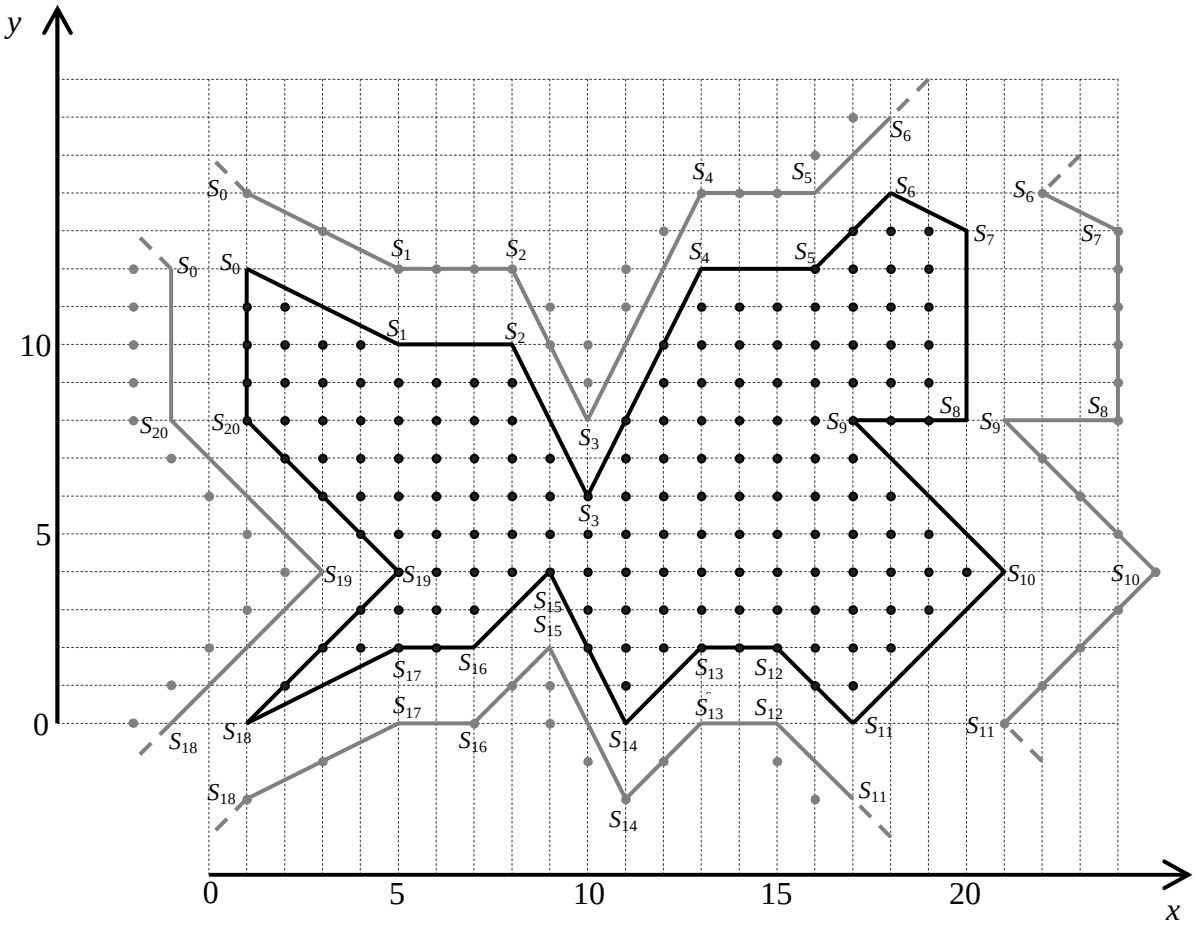


Figure 5 : Remplissage d'un polygone et de ses voisins contigus
(représentés en gris sur la vue éclatée)

Avant de décrire comment sont calculées, en général, les coordonnées des extrémités de ces segments, précisons comment sont traités les deux cas particuliers suivants :

1°) Concernant le point 3 de cet algorithme de principe, un problème est de déterminer, dans le cas où x_{2k} et/ou x_{2k+1} sont entiers, si les pixels d'abscisses x_{2k} et x_{2k+1} doivent ou non être affichés. Ce problème se pose en particulier si on affiche différents polygones contigus avec des couleurs différentes, car il faut éviter que les discrétisations de ces polygones ne se chevauchent.

Un principe permettant d'éviter ces chevauchements est d'afficher les pixels qui se trouvent sur les *arêtes gauches*, *i. e.* les arêtes correspondant aux points d'intersection d'indices pairs, et de ne pas afficher les pixels qui se trouvent sur les *arêtes droites*, *i. e.* les arêtes correspondant aux points d'intersection d'indices impairs.

2°) Un autre problème se pose pour les sommets du polygone, car ils sont à l'intersection de deux arêtes. Doivent-ils ou non former une extrémité d'un segment d'intersection ?

La figure 5 montre l'exemple d'un polygone qui est entouré de polygones voisins contigus. Pour le polygone central, il est naturel que les sommets S_7, S_{10}, S_{19} et S_{20} forment des extrémités de segments d'intersection et que les sommets S_0, S_3, S_6, S_{11}, S_{14}, S_{15} et S_{18} n'en forment pas. Mais la

réponse est moins facile pour des sommets tels que S_1, S_2, S_4, S_5, etc. qui sont des extrémités d'arêtes horizontales.

Un critère qui permet de déterminer si un sommet doit, ou ne doit pas, former une extrémité d'un segment d'intersection est le suivant : si, pour le sommet considéré, le nombre des arêtes non horizontales dont il est le point d'ordonnée minimale est égal à 1 alors le sommet sera une extrémité d'un segment d'intersection, sinon (0 ou 2) il n'en sera pas.

Revenons à l'exemple de la figure 5 :

- Les sommets S_7, S_{10}, S_{19} et S_{20} sont les points d'ordonnée minimale d'une seule arête. Ils sont donc, d'après le critère défini ci-dessus, des extrémités de segments d'intersection. Par exemple, pour $y = 13$, la liste triée du point 2 de l'algorithme est donc (17, 20) (mais il faut noter que le pixel associé à S_7 n'est pas affiché car il appartient à une arête droite).
- Les sommets S_3, S_{11}, S_{14} et S_{18} sont les points d'ordonnée minimale de deux arêtes. Ils ne sont donc pas des extrémités de segments d'intersection. Par exemple, pour $y = 6$, comme S_3 n'est pas une extrémité d'un segment d'intersection, la liste triée est donc (3, 19). Pour $y = 0$, comme aucun des sommets S_{11}, S_{14} et S_{18} ne sont des extrémités de segments d'intersection, la liste reste vide et ces sommets ne sont pas affichés sur le polygone central.
- Les sommets S_0, S_6 et S_{15} ne sont les points d'ordonnée minimale d'aucune arête. Ils ne sont donc pas non plus des extrémités de segments d'intersection. Par exemple, pour $y = 4$, comme S_{15} n'est pas une extrémité d'un segment d'intersection, la liste triée est donc (5, 21).
- Les sommets S_1, S_5, S_8 et S_{16} sont les points d'ordonnée minimale d'une seule arête non horizontale (les arêtes horizontales ne comptent pas). Ils sont donc des extrémités de segments d'intersection. Par exemple, comme S_1 est une extrémité d'un segment d'intersection, pour $y = 10$, le premier segment de la liste triée est donc (1, 5).
- Les sommets S_2, S_4, S_9, S_{12}, S_{13} et S_{17} ne sont les points d'ordonnée minimale d'aucune arête non horizontale. Ils ne sont donc pas des extrémités de segments d'intersection. Par exemple, pour $y = 2$, comme S_{17}, S_{13} et S_{12} ne sont pas des extrémités de segments d'intersection, mais que S_{16} en est un, la liste triée est donc (3, 7, 10, 19).

Comme le montre également la figure 5, ce critère assure que les points à la frontière de polygones contigus appartiennent à l'intérieur d'un seul de ces polygones.

4.2 *Algorithme de parcours des arêtes du polygone*

Revenons au calcul des coordonnées des extrémités des segments d'intersection entre les lignes horizontales et le polygone (point 1 de l'algorithme général). Chaque arête du polygone va d'un pixel (x_{bas}, y_{bas}) à un pixel (x_{haut}, y_{haut}), avec $y_{bas} < y_{haut}$ (puisqu'on ignore les arêtes

horizontales). En posant $\Delta x = x_{haut} - x_{bas}$ et $\Delta y = y_{haut} - y_{bas} > 0$, on a, pour tout point (x, y) de l'arête :

$$x = x_{bas} + \frac{\Delta x}{\Delta y}(y - y_{bas}).$$

En particulier, si $x_i = x_{bas} + \frac{\Delta x}{\Delta y}(y_i - y_{bas})$, et $y_{i+1} = y_i + 1$, alors $x_{i+1} = x_i + \frac{\Delta x}{\Delta y}$.

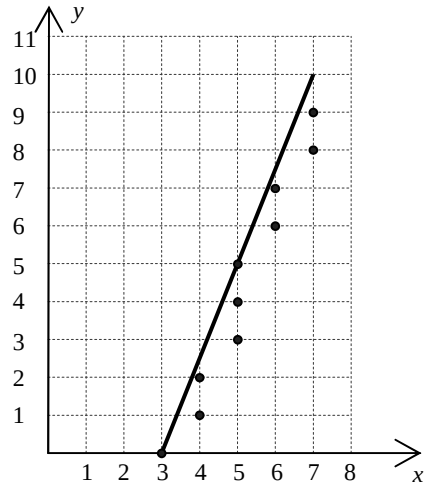
Les abscisses des points d'intersection des arêtes du polygone avec des lignes horizontales d'équation $y = y_i$ pourraient donc être obtenues à partir de la valeur initiale $x = x_{bas}$ en ajoutant, à chaque nouvelle valeur de l'indice i, un incrément égal à $\frac{\Delta x}{\Delta y}$. Cependant, utiliser directement cette relation de récurrence entre nombre réels est trop coûteux pour un affichage rapide, en raison des opérations en virgule flottante et de la nécessaire opération d'arrondi. Les calculs sont donc effectués uniquement en arithmétique entière, avec l'algorithme ci-après (qui prend en entrée les valeurs x_{bas}, y_{bas}, x_{haut} et y_{haut}) :

```
1. x ← x_bas ;
2. △x ← x_haut – x_bas ;
3. △y ← y_haut – y_bas ;
// N représente le numérateur de la partie fractionnaire de x (le dénominateur
étant toujours égal à △y) ; il est initialisé en fonction du type de l'arête
(droite ou gauche) et de sa pente (△x > 0 ou △x ≤ 0) //
4. Si l'arête est une arête gauche, alors
        N ← △y–1
    sinon
        N ← -1
5. De y = y_bas à y_haut – 1, faire :
// Division_euclidienne (A, B)  retourne (Q, R) avec A=BQ+R et 0 ≤ R < |B| //
   5.1.    (Q, N) ← Division_euclidienne (N, △y) ;
   5.2.    x ← x + Q ;
   5.3.    Afficher le pixel (x, y) ;
   5.4.    N ← N + △x.
```

Les différentes initialisations de N permettent de n'afficher que des points à l'intérieur du polygone et d'éviter les chevauchement en affichant les pixels qui se trouvent exactement sur les arêtes gauches, mais pas ceux qui se trouvent sur les arêtes droites. On remarquera également que la boucle 5 ne s'exécute que jusqu'à $y = y_{haut} - 1$ car il a été convenu dans la section précédente de ne considérer, pour chaque sommet du polygone, que l'arête dont il est le point d'ordonnée minimale

L'exemple ci-après illustre le cas d'une arête gauche à pente positive. Prenons $(x_{bas}, y_{bas}) = (3, 0)$ et $(x_{haut}, y_{haut}) = (7, 10)$. On a $\Delta x = 4$, $\Delta y = 10$, et N initialisé à $\Delta y - 1 = 9$. L'initialisation de N à 9 revient à "décaler" la valeur réelle de x de 0,9, de façon à ce que l'arrondi effectué par la division euclidienne affiche x quand x est entier et x+1 quand il ne l'est pas, ce qui satisfait bien les propriétés souhaitées pour une arête gauche, comme le montre la figure 6.

Le tableau donne respectivement, pour chaque valeur de *y*, les valeurs de *Q* et de *N* à la ligne 5.1, la valeur réelle de *x*, sa valeur “décalée” de 0,9, la valeur de la variable *x* qui est affichée et la valeur de *N* à la ligne 5.4.



y	(*Q*, *N*)	$x_{réel}$	$x_{décalé}$	$x_{affiché}$	*N*
0	(0, 9)	3	3,9	3	13
1	(1, 3)	3,4	4,3	4	7
2	(0, 7)	3,8	4,7	4	11
3	(1, 1)	4,2	5,1	5	5
4	(0, 5)	4,6	5,5	5	9
5	(0, 9)	5	5,9	5	13

Figure 6 : Parcours d’une arête gauche de pente positive

4.3 *Procédure de remplissage d’un polygone*

Dans cette procédure, les sommets du polygone sont représentés par des variables appartenant au type structuré `sommet`. Ce type est dit structuré car il comporte deux champs `x` et `y`, de type entier, représentant respectivement l’abscisse et l’ordonnée du sommet représenté par la variable. Par exemple, pour une variable `S` de type `sommet`, le champ `x` de `S`, noté `S.x`, est un entier qui représente l’abscisse du sommet représenté par la variable `S`. Un polygone de sommets S_0, …, S_{n-1} est alors représenté par un tableau d’éléments de type `sommet`, `TS[0..n-1]`, `TS[`*i*`]` représentant le sommet S_i.

De même, les arêtes du polygone sont représentées par des variables appartenant au type structuré `arête`, qui comporte quatre champs `xbas`, `ybas`, `xhaut`, `yhaut`, eux-mêmes de type entier.

L’idée générale de la procédure est la suivante :

1. une procédure `Construire_TAG_et_TAD` construit deux tableaux d’arêtes, `TAG` et `TAD`, contenant respectivement les arêtes gauches et droites du polygone ;
2. pour chacune de ces arêtes et pour chaque valeur de *y* entre `ybas` et `yhaut`, une procédure `Parcours`, basée sur l’algorithme décrit dans la section précédente, détermine l’abscisse de l’extrémité du segment horizontal d’ordonnée *y* inclus dans le polygone. Si l’arête est une arête gauche (respectivement droite), cette abscisse est insérée dans un tableau `XG` (respectivement `XD`) ;
3. Les tableaux `XG` et `XD` sont utilisés pour effectuer l’affichage des points intérieurs du polygone.

Le principe de la procédure `Construire_TAG_et_TAD`, du point 1, est le suivant : si, quand on parcourt une arête en tournant autour du polygone dans le sens trigonométrique inverse l'ordonnée augmente, alors l'arête est une arête gauche, sinon, c'est une arête droite. La procédure, qui prend en entrée un polygone, représenté par un tableau de sommets triés dans le sens trigonométrique inverse, s'écrit donc ainsi :

```
Procédure Construire_TAG_et_TAD (Tableau de sommets : TS[0..n–1])
// Les sommets de TS sont ordonnés dans le sens trigonométrique inverse //
Entiers i, jg, jd, imin ;
   1. Calcul de ymin, ymax et imin, indice du premier sommet d'ordonnée ymin ;
   2. i ← imin ; jg ← 0 ; jd ← 0 ;
   3. Répéter :
        a.  Si i < n-1 alors isuivant ← i+1 sinon isuivant ← 0 ;// i+1 modulo n-1 //
        b.  Si TS[isuivant].y > TS[i].y alors                  // arête gauche //
             i.   TAG[jg].xbas ← TS[i].x ;
            ii.   TAG[jg].ybas ← TS[i].y ;
           iii.   TAG[jg].xhaut ← TS[isuivant].x ;
            iv.   TAG[jg].yhaut ← TS[isuivant].y ;
             v.   jg ← jg+1 ;
        c.  Si TS[isuivant].y < TS[i].y alors                  // arête droite //
             i.   TAD[jd].xbas ← TS[isuivant].x ;
            ii.   TAD[jd].ybas ← TS[isuivant].y ;
           iii.   TAD[jd].xhaut ← TS[i].x ;
            iv.   TAD[jd].yhaut ← TS[i].y ;
             v.   jd ← jd+1 ;
        d.  i ← isuivant ;
  jusqu'à ce que i=imin ;
   4. Nombre_AG ← jg ; Nombre_AD ← jd ;
```

On notera que cette procédure calcule également diverses variables utilisées ultérieurement : `ymin` et `ymax`, ordonnées minimale et maximale des sommets du polygone ; `imin`, indice du premier sommet d'ordonnée `ymin` ; `Nombre_AG` et `Nombre_AD`, nombres d'arêtes gauches et droites du polygone.

Comme le suggère le point 2, l'intersection d'une horizontale avec le polygone peut être constituée par plusieurs segments et les abscisses x_0, …, x_{2n-1} de leurs extrémités sont rangées dans deux tableaux bidimensionnels `XG` et `XD` (à `NbLig` lignes et `NbCol` colonnes) tels que, pour $k \in [\![0, \texttt{NbCol} - 1]\!]$, `XG`$[y,k]$ et `XD`$[y,k]$ contiennent respectivement les abscisses x_{2k} et x_{2k+1} des extrémités du $(k+1)^{\text{ème}}$ segment d'intersection du polygone et d'une droite horizontale à l'ordonnée y. Les lignes des tableaux `XG` et `XD` doivent donc rester triées par ordre d'abscisses croissantes.

La procédure `Parcours` qui permet de remplir ces deux tableaux est la suivante :

```
Procédure Parcours (Entiers : xbas, ybas, xhaut, yhaut, type_arête)
Entiers : N, x, y, ∆x, ∆y ;
1   x ← xbas ;
2   ∆x ← xhaut – xbas ;
3   ∆y ← yhaut – ybas ;
4   Si type_arête = gauche, alors
         N ← ∆y–1
    sinon
         N ← -1
5   De y = ybas à yhaut - 1, faire :
  5.1  (Q, N) ← Division_euclidienne (N, ∆y) ;
  5.2  x ← x + Q ;
  5.3  Si type_arête = gauche, alors insérer x dans la ligne XG[y]
                               sinon insérer x dans la ligne XD[y]
  5.4  N ← N + ∆x.
```

La procédure de remplissage du polygone est donc finalement :

```
Procédure Remplissage (Tableau de sommets : TS[0..n–1])
Entiers i, k, x, y, ymin, ymax ; Nombre_AG, Nombre_AD ;
Tableaux d'arêtes : TAG[0..Taille_TAG-1], TAD[0..Taille_TAD-1] ;
Tableaux d'entiers : XG[0..NbLig-1,0..NbCol-1], XD[0..NbLig-1,0..NbCol-1] ;
1   Initialiser XG et XD avec des -1 ;    //les abscisses sur l'écran sont >= 0//
2   Construire_TAG_et_TAD(TS) ;
3   Trier les tableaux TAG et TAD suivant les valeurs croissantes de xbas ;
4   De i=0 à Nombre_AG - 1, faire
    Parcours(TAG[i].xbas, TAG[i].ybas, TAG[i].xhaut, TAG[i].yhaut, gauche) ;
5   De i=0 à Nombre_AD - 1, faire
    Parcours(TAD[i].xbas, TAD[i].ybas, TAD[i].xhaut, TAD[i].yhaut, droite) ;
6   De y=ymin à ymax - 1  faire
    6.1 k ← 0 ;
    6.2 Répéter
      6.2.1 De x=XG[y,k] à XD[y,k] Afficher(x,y) ;
      6.2.2 k ← k+1
      jusqu'à ce que XG[y,k]=-1.
```

On remarquera que le tri des tableaux `TAG` et `TAD` suivant les valeurs croissantes de `xbas` permet de faire que, souvent, la valeur `x` sera insérée à la fin de la ligne, mais que ceci n'est pas assuré de manière systématique.

Le document fourni en annexe donne un exemple d'exécution de la procédure de remplissage pour le polygone de la figure 5.

5. Conclusion

Pour replacer l'algorithme de remplissage dans le contexte 3D de l'algorithme du *z-buffer*, il suffit, dans le point 6.2.1 d'adapter la procédure `Afficher(x,y)` en utilisant la procédure de traitement qui correspond au point 3.3 de l'algorithme de principe donné en page 3 :

```
Procédure Afficher (Entiers : x, y)
1 Calculer le point P de Π qui se projette sur (x,y) ;
2 Si P.z > 0 et P.z < Min[x,y], alors
  2.1 Min[x,y] ← P.z
  2.2 Mettre la couleur du point P sur le pixel (x,y).
```

Document annexe : remplissage d'un polygone

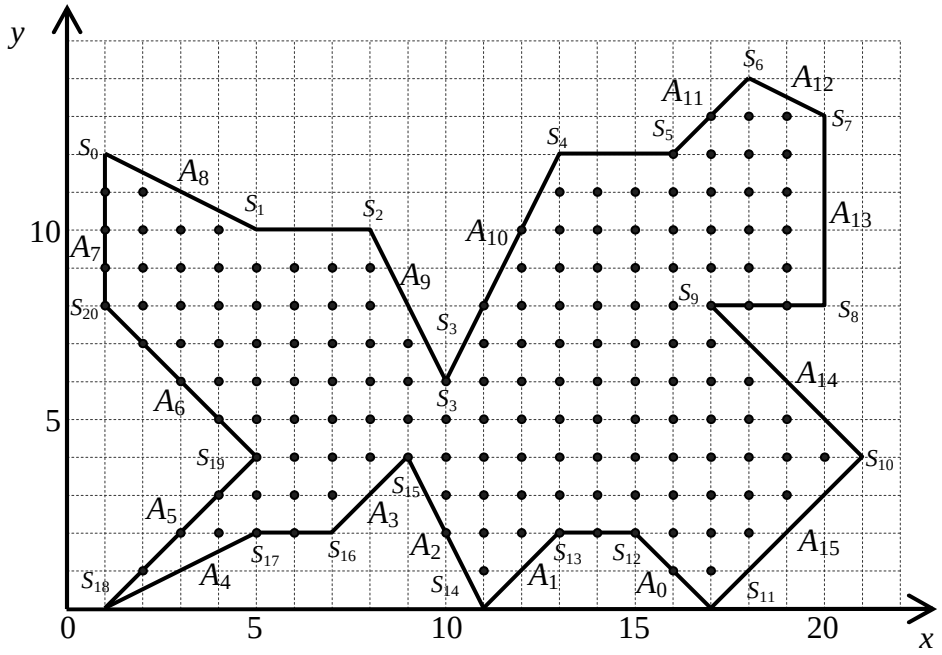


Figure 8 : Remplissage d'un polygone

On a `imin` = 11, `ymin` = 0 et `ymax` = 14.

Les tableaux d'arêtes correspondant au polygone ci-dessus sont :

`TAG`

	xbas	ybas	xhaut	yhaut
A_0	17	0	15	2
A_2	11	0	9	4
A_5	1	0	5	4
A_6	5	4	1	8
A_7	1	8	1	12
A_{10}	10	6	13	12
A_{11}	16	12	18	14

`TAD`

	xbas	ybas	xhaut	yhaut
A_1	11	0	13	2
A_3	7	2	9	4
A_4	1	0	5	2
A_8	5	10	1	12
A_9	10	6	8	10
A_{12}	20	13	18	14
A_{13}	20	8	20	13
A_{14}	21	4	17	8
A_{15}	17	0	21	4

Une fois triés suivant les valeurs croissantes de `xbas`, ces tableaux deviennent :

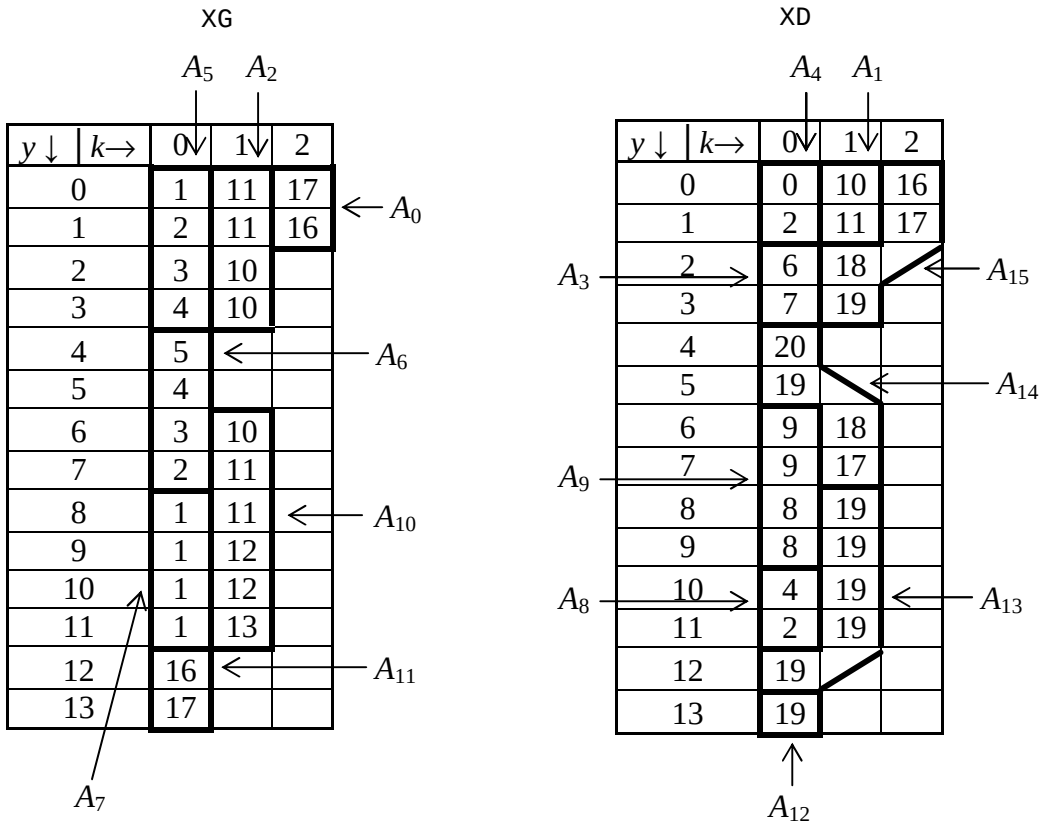
`TAG`

	xbas	ybas	xhaut	yhaut
A_5	1	0	5	4
A_7	1	8	1	12
A_6	5	4	1	8
A_{10}	10	6	13	12
A_2	11	0	9	4
A_{11}	16	12	18	14
A_0	17	0	15	2

`TAD`

	xbas	ybas	xhaut	yhaut
A_4	1	0	5	2
A_8	5	10	1	12
A_3	7	2	9	4
A_9	10	6	8	10
A_1	11	0	13	2
A_{15}	17	0	21	4
A_{13}	20	8	20	13
A_{12}	20	13	18	14
A_{14}	21	4	17	8

Les tableaux `XG` et `XD` obtenus avec les appels à la procédure `Parcours` sont :



On remarquera qu'en affichant les pixels entre `XG[`y`,`k`]` et `XD[`y`,`k`]`, pour y variant de 0 à 13 et k variant de 0 à 0 ou 1 ou 2 selon les lignes, on obtient bien les pixels affichés à l'intérieur du polygone de la figure 8. Par exemple : comme `XG[0,0]` > `XD[0,0]`, `XG[0,1]` > `XD[0,1]` et `XG[0,2]` > `XD[0,2]`, aucun pixel n'est affiché sur la ligne $y = 0$; sur la ligne $y = 6$, on trouve deux segments contigus $[\![3, 9]\!]$ et $[\![10, 18]\!]$, donc tous les pixels entre $x = 3$ et $x = 18$ sont affichés.