Patrick TRAU
http://pat.fr.st

Programmer en C (suite)

Données et Algorithmes

Algorithmes d'infographie

Université Louis Pasteur
Institut Professionnel des
Sciences et Technologies
15 rue du Maréchal Lefèbvre
67100 STRASBOURG (F)

Table des matières

Table des matières

Table des matières

Table des matières

1 Données et Algorithmique

Ce document décrit les structures de données et les algorithmes que l'on peut leur associer. Contrairement à beaucoup d'ouvrages d'algorithmique, j'ai préféré mettre l'accent sur le choix des structures de données. Il est avant tout important de définir coment modéliser le problème à traiter, ainsi que ses données. Ce choix fait, on cherchera ensuite l'algorithme optimal, adapté aux types de données choisies. Ce document n'est pas spécifique à un langage (il suffit qu'il soit structuré). Les exemples par contre sont tous donnés en C (ANSI), mais sont facilement transposables dans un autre langage. Vous trouverez les informations nécessaires dans mon document sur le langage C ou, pour un détail particulier, son index.

Autres sites sur l'algorithmique : Jean−Jacques Levy (Polytechnique), Jean Beuneu (EUDIL Lille), cours de JL Bienvenu (CNAM Bordeaux), les polycopiés de Jean Fruitet (IUT Marne la Vallée)

---

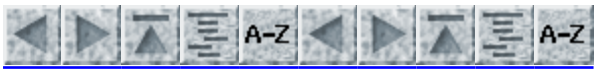
2 INTRODUCTION

Lorsque l'on désire créer un programme répondant à un cahier des charges bien défini (condition préalable évidement nécessaire), il faut déterminer quelles données il va falloir traiter, et comment les traiter. La première étape est donc de choisir comment représenter en mémoire ces données, et si plusieurs possibilités sont envisageables, choisir la plus appropriée aux traitements qu'il faudra effectuer (c'est à dire celle pour laquelle les algorithmes seront le plus facile à mettre en oeuvre). Dans un gros programme (C.A.O. par exemple), on appelle *modèle* la structure choisie pour stocker les données en mémoire.

Une fois ce modèle défini, le programme doit être écrit de manière *structurée*, c'est à dire être décomposé en petites entités (sous programmes, fonctions en C), réalisant chacune une tâche bien définie, en ayant bien défini quelles sont les données nécessaires en entrée du sous programme, et quelles seront les données retournées en sortie du sous programme (arguments ou dans certains cas variables globales). La réalisation pratique de la tâche doit ne dépendre que de ses entrées et sorties, et n'accéder à aucune autre variable (par contre elle peut utiliser pour son propre compte autant de variables locales que nécessaire). Ceci permet d'éviter les *effets de bord*, qui rendent la recherche d'erreurs (débogage) presque impossible.

Le choix d'un modèle est capital : devoir le modifier une fois le programme bien avancé nécessite en général la réécriture complète du programme, alors que modifier certaines fonctionnalités du programme correspond à ajouter ou modifier des sous programmes sans modifier les autres. C'est un des intérêts de la programmation structurée. Par contre, pour pouvoir plus facilement modifier le modèle, il faut des structures de données hiérarchisées et évolutives (disponibles dans les *langages orientés objets*). Un autre avantage de la programmation structurée est la possibilité de créer dans un premier temps chaque sous programme réalisant une tâche déterminée grâce à un algorithme simple, puis d'optimiser uniquement les sous−programmes souvent utilisés, ou demandant trop de temps de calcul, ou nécessitant trop de mémoire.

Parlons encore de l'*optimisation* d'un programme. On n'optimise un programme (ou du moins certaines parties) que si l'on estime que son fonctionnement n'est pas acceptable (en temps ou en consommation de mémoire). On devra choisir un algorithme en fonction des conditions d'utilisation du programme (on ne trie pas de la même manière un fichier totalement mélangé et un fichier déjà trié , mais avec quelques valeurs non triées en fin de fichier). A partir d'un moment, on ne peut plus optimiser en temps et en mémoire. Il faut alors choisir. Par exemple, un résultat de calcul qui doit être réutilisé plus tard peut être mémorisé (gain de temps) ou on peut préférer refaire le calcul (gain de mémoire). Par exemple, il est rare de passer par une variable intermédiaire pour utiliser deux fois i+1.



3 LES VARIABLES SCALAIRES

- codage binaire
- boucles
- récursivité

3.1 codage binaire

Les seules valeurs que peut traiter l'ordinateur (et d'ailleurs tout système numérique) est le 0 et le 1 (en fait, c'est nous qui représentons par 0 ou 1 le fait que le système numérique ait vu du courant ou non). Un problème important est que différents types d'informations doivent être codés : instructions machine (lignes de programmes), valeurs numériques, caractères, adresses,... et que rien ne permet de distinguer dans un groupe de 0 et de 1 le type d'information qu'il est censé représenter (10010110 est sur un PC le code machine permettant d'échanger les contenus des registres SI et AX, mais également le code du caractère 'û', de l'octet signé décimal −106, du signé 150, l'exposant du flottant $1{,}23x10^{-32}$, etc...). Il est donc capital d'associer à chaque mémoire que l'on utilise le type de donnée que l'on y mettra. C'est ce que permet la déclaration des variables dans les langages évolués (explicite en Pascal et en C, implicite en Basic et Fortran). Néanmoins cette erreur reste possible (en C, lors d'une erreur sur des pointeurs, par l'utilisation des unions, ou simplement par erreur de format dans un printf).

Une autre erreur est due à la représentation des nombres négatifs. En effet, le signe ne peut être, lui aussi, représenté que par 0 ou 1. Le codage choisi (pour les entiers 16 bits) fait que l'ajout de 1 à 32767 (le plus grand entier signé) donne −32768 (cela devait de toute façon donner un nombre car soit il y a du courant, soit il n'y en a pas, il n'est pas prévu de combinaison de 0 et 1 représentant une erreur). La plupart des compilateurs ne signalent pas d'erreur en cas de dépassement de capacité de nombres entiers.

Autre problème, la codification des réels. Représenter la présence ou non d'une virgule par un 0 ou un 1 est évidement impossible (comment la reconnaître ?), la première solution envisagée était donc la virgule fixe (un mot pour la partie entière, un autre pour la partie fractionnaire). On utilise désormais la "virgule flottante" (d'où le nom de flottants ou float), représentée par un mot pour la mantisse (qui est un réel en virgule fixe puisque pas de partie entière), un autre (souvent de taille différente) pour l'exposant (entier signé). Ceci implique deux limitations : le nombre de chiffres significatifs (dû à la taille de la mantisse), et le plus grand réel codifiable (dû à la taille de l'exposant, par exemple $2^{127}=1{,}7.10^{38}$). On cherchera donc à ne combiner que des réels du même ordre de grandeur. Par exemple en mécanique, ajouter à une dimension d'un mètre une dilatation thermique d'un micron n'a de sens que si les flottants possèdent plus de 6 chiffres significatifs, donc par exemple un algorithme cumulatif ne donnera pas de résultat si le pas en température est trop faible.

Dans les deux cas (virgule fixe ou flottante), un problème se pose : en fait, nous ne pouvons représenter qu'un sous ensemble des réels (je ne pense pas que les mathématiciens lui aient donné un nom) qui correspond à la différence, en base 10, entre D et R. En fait on ne peut représenter que les nombres pouvant s'écrire sous forme d'une partie fractionnaire comportant un nombre fini de chiffres (en binaire). Or, comme c'est impossible en base 10 pour 1/3 (qui pourtant s'écrit 0,1 en base 3) la représentation en binaire de 1/10 donne une suite infinie, qui est donc toujours tronquée ($0{,}1_d$ = $0{,}00010001001001001001..._b$). donc sur tout ordinateur calculant en binaire, (1/10)*10 donne 0,999999999... Cette erreur devient gênante dans le cas où le résultat du calcul précédent est utilisé dans le calcul suivant, pour de grandes suites de calculs.

exemple (flottant):

```
#include <stdio.h>
void main(void)
 {
  float x=1000;
```

```
  int i;
  for (i=0;i<10000;i++)x+=0.1;
  printf("On obtient %12.4f au lieu de 2000.0000\n",x);
 }
```

Ce problème est moins flagrant en C que dans les autres langages, le C effectuant toujours les calculs sur des réels en double précision. Il en résulte néanmoins que, par exemple, il ne faut pas tester dans un programme si le résultat d'un calcul flottant est nul mais si sa valeur absolue est inférieure à un petit nombre. On cherchera aussi, autant que possible, à choisir des algorithmes utilisant des entiers plutôt que des réels, d'autant plus que les calculs sur des flottants sont plus lents que sur des réels.

3.2 boucles

Tous les langages possèdent des structures permettant de répéter des instructions, les boucles. En général, certaines sont réservées aux cas où l'on connaît à l'entrée le nombre de boucles à effectuer (for en C), et celles dont on connaît la condition d'arrêt (while et do−while en C). (voir exemples dans la première partie).

Lorsqu'une valeur à calculer C dépend de n calculs intermédiaires C_i (que l'on ne désire pas mémoriser), la méthode la plus simple consiste à initialiser C à la première valeur C_1, puis pour i variant de 2 à n, calculer chaque C_i et le cumuler à C. C'est en général la méthode utilisée pour les calculs de sommes, produits, suites...

exemple (boucle_A) : calculer x^n :

```
#include <stdio.h>
void main(void)
 {
  int n,i,x,result;
  printf("entrez x et n : ");
  scanf("%d %d",&x,&n);
  result=x;
  for(i=1;i<n;i++)result*=x;
  printf("résultat : %d\n",result);
 }
```

La fonction puissance est souvent disponible dans les langages (pow en C), mais correspond à un calcul assez long et souvent moins précis ($x^n=\exp(n*\ln(x))$ qui donne toujours un résultat avec une erreur de précision même si x est entier). La méthode ci−dessus sera plus rapide soit pour n petit (quelques multiplications d'entiers sont plus rapides qu'une exponentielle de logarithme), soit quand toutes les puissances intermédiaires doivent être connues.

Exercice (boucle) : faire le programme calculant $2x^4-3x^3+x^2-5x+2$ pour x réel.

exemple (boucle_B) : calcul de moyenne :

```
#include <stdio.h>
void main(void)
 {
  int n=0;
  float note,moy=0;
  do
   {
    printf("entrez la note (négative pour terminer) : ");
    scanf("%f",&note);
    if(note>=0)
     {
      moy+=note;
      n++;
     }
   }
```

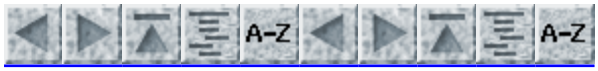
```
  while(note>=0);
  moy/=n;
  printf("moyenne : %f\n",moy);
 }
```

Les valeurs des notes ayant servi au calcul ne sont pas mémorisées, car toutes stockées successivement dans la même mémoire.

3.3 récursivité

On appelle fonction *récursive* une fonction qui s'appelle elle−même. La récursivité n'est possible que dans un langage acceptant des variables locales. En effet, l'emploi de la récursivité n'est réellement utile que lorsqu'une fonction doit retrouver, après un appel récursif, toutes ses variables locales dans leur état initial. Dans le cas contraire, une méthode itérative (boucle) sera en général plus efficace (il est inutile de mémoriser les variables locales et les restaurer pour ne plus les réutiliser), comme par exemple pour le calcul d'une factorielle (voir mon document sur le C, paragraphes 4.6 et 4.7, pour l'explication de la récursivité, l'empilement des variables locales et l'exemple de la factorielle).

Néanmoins dans un certain nombre de cas, l'emploi de la récursivité facilite la programmation. Si le temps de calcul ou la mémoire utilisée sont prohibitifs pour de grands nombres de données, il est *toujours* possible de traduire un algorithme récursif en itératif, et souvent de manière plus efficace que le compilateur, qui est obligé de le faire (le langage machine n'est pas récursif) sans connaître aussi bien que vous les conditions d'utilisation de l'algorithme.



4 LES TABLEAUX STATIQUES

4.1 tableaux unidimensionnels

4.1.1 généralités

Un tableau permet de regrouper dans une structure plusieurs valeurs scalaires *de même type*. Pour permettre une maintenance aisée du programme, la dimension doit être définie par une constante. C'est cette dernière qui sera utilisée pour les tests de dépassement (peu de compilateurs le font automatiquement, ils le feraient par exemple à l'intérieur d'une boucle alors que le test sur la valeur finale serait suffisant). La taille réellement utilisée peut être inférieure ou égale à la dimension du tableau, les composantes au delà de la taille utilisée peuvent être mises à 0 mais cela n'a aucun intérêt, sauf si on interdit le 0 dans le tableau (cas des caractères, en C).

Nous allons travailler dans un premier temps sur des tableaux de flottants. Nous utiliserons des tableaux dont le premier indice est 0 puisque c'est la seule possibilité en C, et la solution optimale dans les autres langages puisque nécessitant moins de calculs dans le code compilé. Si tous les tableaux utilisés ont la même dimension, on peut définir de manière globale :

```
#define composante float
#define dim_tus 100
typedef composante type_tus[dim_tus];
```

Ces déclarations sont nécessaires dans les langages effectuant un contrôle très strict des types de données (comme le Pascal) et refusant de combiner des tableaux de tailles différentes. En C, nous pouvons préparer des fonctions sur tous tableaux de réels, quelle que soit leur dimension :

```
#define composante float
typedef composante type_tus[]; /* ou typedef composante *type_tus */
```

C'est cette seconde déclaration que nous utiliserons dans les exemples suivants.

Remarque : dans notre cas il peut être intéressant de définir un tableau comme une structure regroupant le tableau, sa dimension et sa taille effective.

4.1.2 fonctions de base (fichier inclus base_tus)

Les fonctions de base sur les tableaux unidimensionnels sont l'initialisation du tableau, l'ajout d'une valeur en fin du tableau, et l'affichage de tout le tableau :

```
void init_tus(type_tus tab,int *taille,int dim)
 {
  int i;
  for(i=0;i<dim;i++) tab[i]=0;
  *taille=0;
 }
```

L'initialisation à 0 n'est pas nécessaire dans la plupart des cas. La taille est passée par adresse puisque modifiée par la fonction init_tus.

```
int ajoute_val_tus(type_tus tab,int *taille,int dim,composante val)
/* retourne 0 si tout s'est bien passé, 1 si erreur */
 {
  if (*taille>=dim)
   {
    puts("dépassement de capacité du tableau");
    return(1);
   }
  /* le else n'est pas nécessaire du fait du return précédent */
  tab[(*taille)++]=val;
  return(0);
 }

void affiche_tus(type_tus tab,int taille)
 {
  int i;
  for(i=0;i<taille;i++) printf("%dième valeur :
%f\n",i+1,tab[i]);
 }
```

Les déclarations globales et fonctions ci−dessus étant recopiées dans le fichier "base_tus", nous pouvons écrire le programme suivant (ex_tus) :

```
#include <stdio.h>
#include "base_tus.inc"
#define dim 100
void main(void)
 {
  int nb;
  composante v,t[dim];
  init_tus(t,&nb,dim);
  do
   {
    printf("entrez la %dième note (fin si <0 ou >20) :",nb+1);
    scanf("%f",&v);
    if(v<0||v>20) break; /* on aurait pu le mettre
                                dans la condition du while */
   }
  while (!ajoute_val_tus(t,&nb,dim,v));
  affiche_tus(t,nb);
 }
```

Exercice (tusexo_a) : modifier ce programme pour qu'il calcule la moyenne et affiche, pour chaque note, l'écart avec la moyenne (ce qui nécessite l'utilisation d'un tableau car il faut d'abord calculer la moyenne puis

utiliser les notes mémorisées auparavant).

4.1.3 manipulations dans les tableaux (mani_tus)

L'insertion et la suppression de composantes d'un tableau nécessitent des décalages des autres composantes du tableau:

```
void suppr_tus(type_tus tab,int *taille,int position)
 {
  int i;
  if(position>=*taille||position<0)return;
  (*taille)−−;
  for(i=position;i<*taille;i++)tab[i]=tab[i+1];
 }
```

On peut remarquer que la suppression de la dernière composante n'entraîne aucun décalage.

```
int insert_tus(type_tus tab,int *taille,int dim,int position,composante val)
/* retourne 0 si pas d'erreur */
 {
  int i;
  if(position<0)return(1);
  if(position>*taille)position=*taille;
  if (*taille>=dim)
   {
    puts("dépassement de capacité du tableau");
    return(2);
   }
  for(i=*taille;i>position;i−−)tab[i]=tab[i−1];
  tab[position]=val;
  (*taille)++;
  return(0);
 }
```

Le décalage doit se faire par indice décroissant (par indice croissant, on recopierait progressivement la composante à l'indice position dans tout le reste du tableau).

Les rotations sont également des manipulations fréquentes sur les tableaux :

```
void rot_gauche_tus(type_tus tab,int taille)
 {
  composante tampon;
  int i;
  tampon=tab[0];
  for(i=1;i<taille;i++)tab[i−1]=tab[i];
  tab[taille−1]=tampon;
 }
void rot_droite_tus(type_tus tab,int taille)
 {
  composante tampon;
  int i;
  tampon=tab[taille−1];
  for(i=taille−1;i>0;i−−)tab[i]=tab[i−1];
  tab[0]=tampon;
 }
```

Exercice (tusexo_b) : faire un programme permettant de tester ces fonctions, à l'aide d'un menu permettant d'essayer dans n'importe quel ordre ces fonctions.

4.1.4 tris

4.1.4.1 généralités (valables également pour d'autres types de données)

Les exemples qui suivent traitent des tableaux de flottants, les méthodes étant identiques pour tout type de composante à condition d'y définir une relation d'ordre (par exemple, pour des chaînes de caractères on utilisera *strcmp* au lieu de <, > et =). Mais il ne faut pas oublier que l'efficacité de l'algorithme dépend également des types de données traitées : une comparaison de chaînes de caractères étant relativement longue, les algorithmes effectuant beaucoup de tests seront moins efficaces qu'avec des flottants. De même, les tableaux trop gros pour entrer en mémoire devront être traités sur support externe, rendant l'accès aux données (plusieurs millisecondes) bien plus lent que les tests ou calculs (micro voire nanosecondes). Dans les cas plus complexes, comme par exemple les tableaux de structures, on appelle *clef* le champ servant pour le tri (par exemple le nom pour un tableau contenant nom, prénom, adresse,...). Dans ce cas, les calculs sur les clefs seront souvent plus rapides que les déplacements des structures entières. Les méthodes de tris présentées ici sont souvent utilisables également avec les autres types de données, mais les conclusions sur leur efficacité varieront. On qualifiera de *stable* un tri laissant dans le l'ordre initial les éléments de clef identique (par exemple, un stock saisi au fur et à mesure des arrivées de matériel, le classement par ordre alphabétique gardant les matériels de même nom dans leur ordre d'arrivée sera stable). Tous les algorithmes décrits ici sont stables, à condition d'y prendre garde (scruter le tableau du début vers la fin et non l'inverse par exemple). Dans ce chapitre, nous noterons N le nombre de composantes du tableau (appelé *taille* auparavant).

4.1.4.2 le tri bulle

Cet algorithme est relativement connu, bien qu'il soit rarement efficace (en termes de temps de calcul, le tri est néanmoins correct, bien évidement). Il consiste à balayer tout le tableau, en comparant les éléments adjacents et les échangeant s'ils ne sont pas dans le bon ordre. Un seul passage ne déplacera un élément donné que d'une position, mais en répétant le processus jusqu'à ce plus aucun échange ne soit nécessaire, le tableau sera trié. (bull_tus)

```
void tri_bulle_tus(type_tus tab, int N)
 {
  int ok,i;
  composante tampon;
  do
   {
    ok=1; /* vrai */
    for(i=1;i<N;i++) if(tab[i−1]>tab[i])
     {
      ok=0;
      tampon=tab[i−1];
      tab[i−1]=tab[i];
      tab[i]=tampon;
     }
   }
  while(!ok);
 }
```

Ce tri va nécessiter un grand nombre de déplacements d'éléments, il est donc inutilisable dans les cas où ces déplacements sont coûteux en temps. Il va nécessiter N−1 boucles principales dans le cas où le dernier élément doit être placé en premier. Le nombre de boucles internes maximal est donc de l'ordre de $(N-1)^2$. Il peut par contre être intéressant quand le tableau initial est déjà pré−trié, les éléments n'étant pas disposés trop loin de leur position finale (par exemple classement alphabétique où les éléments sont déjà triés par leur première lettre).

Plutôt que de déplacer un élément dans une position meilleure que la précédente mais néanmoins mauvaise, les deux algorithmes qui suivent tentent de déplacer les éléments directement en bonne position.

4.1.4.3 le tri par insertion

Plutôt que de déplacer les éléments d'une position, on peut prendre un élément après l'autre dans l'ordre initial, et le placer correctement dans les éléments précédents déjà triés, comme on le fait lorsque l'on classe ses cartes à jouer après la donne (inse_tus) :

```
void tri_insertion_tus(type_tus tab, int N)
 {
  int pt,ppg; /* position testée, premier plus grand */
  composante tampon;
  for(pt=1;pt<N;pt++)
   {
    ppg=0;
    while(tab[ppg]<=tab[pt]&&ppg<pt)ppg++;
    if(ppg<pt)
     {
      tampon=tab[pt];
      suppr_tus(tab,&N,pt);
      insert_tus(tab,&N,32767,ppg,tampon); /* je suis sur de ne pas
              dépasser la dimension puisque je viens de supprimer un
              élément */
     }
   }
 }
```

Le nombre maximal de boucles internes est descendu à N(N−1)/2, on a au maximum N−1 couples de suppression/insertion mais qui eux effectuent en moyenne environ N/2 échanges d'éléments, ce qui amène un maximum d'échanges de l'ordre de N^2, ce qui n'est pas satisfaisant. On peut néanmoins améliorer l'implantation de cet algorithme en optimisant la recherche de la position d'un élément dans la partie déjà triée, par dichotomie par exemple (voir chapitre sur les recherches), ainsi qu'en améliorant le couple suppression/insertion (ne décaler que les éléments entre la position finale et la position initiale, par une rotation à droite). On remplace la boucle principale par :

```
  for(pt=1;pt<N;pt++)
   {
    dpg=pt−1;
    tampon=tab[pt];
    while(tab[dpg]>tampon&&dpg>=0)
       {tab[dpg+1]=tab[dpg];dpg−−;}
    tab[dpg+1]=tampon;
   }
```

Le tri par insertion peut être intéressant pour des tableaux ayant déjà été triés, mais où l'on a rajouté quelques nouveaux éléments en fin de tableau (dans ce cas il faut améliorer l'implantation pour découvrir rapidement le premier élément mal placé, puis utiliser l'algorithme complet pour les éléments restants). Dans les autres cas, il sera plutôt réservé aux types de données permettant une insertion rapide (listes chaînées par exemple).

4.1.4.4 le tri par sélection

Le but est désormais de déplacer chaque élément à sa position définitive. On recherche l'élément le plus petit. Il faut donc le placer en premier. Or cette position est déjà occupée, on se propose donc d'échanger les deux éléments. Il ne reste plus qu'à répéter l'opération N fois (sele_tus):

```
void tri_selection_tus(type_tus tab, int N)
 {
  int pd,pp,i; /* place définitive, plus petit */
  composante tampon;
  for(pd=0;pd<N−1;pd++)
   {
    pp=pd;
```

```
    for(i=pp+1;i<N;i++) if(tab[i]<tab[pp])pp=i;
    tampon=tab[pp];
    tab[pp]=tab[pd];
    tab[pd]=tampon;
   }
 }
```

Chaque échange met un élément en position définitive, l'autre par contre est mal placé. Mais aucun échange n'est inutile. Un élément qui a été bien placé ne sera plus testé par la suite. Le nombre de boucles internes est environ N(N−1)/2, ce qui est meilleur que le tri bulle, mais toujours de l'ordre de N^2. Par contre le nombre de déplacements d'éléments est au maximum de 2(N−1), la moitié étant des déplacements nécessaires, ce qui est faible pour un fichier en désordre total, mais n'est pas optimal pour les fichiers dont la première partie est déjà classée (et grande par rapport à la taille totale). Une amélioration possible serait d'essayer de placer le second élément de l'échange dans une position pas trop mauvaise, à condition que cette recherche ne soit pas elle même plus gourmande en temps.

4.1.4.5 le tri shell

C'est une amélioration du tri par insertion : au lieu d'effectuer une rotation de tous les éléments entre la position initiale et finale (ou du moins meilleure) d'un élément, on peut faire des rotations par pas de P, ce qui rendra le fichier presque trié (chaque élément sera à moins de P positions de sa position exacte). On répète ce tri pour P diminuant jusqu'à 1. Une suite possible pour P est de finir par 1, les pas précédents étant de 4, 13, 40, 121, 364, 1093... ($P_i=3*P_{i-1}+1$). D'autres suites sont évidement possibles, à condition de prendre des valeurs qui ne soient pas multiples entre elles (pour ne pas toujours traiter les mêmes éléments et laisser de côté les autres, par exemple les puissances successives de 2 ne traiteraient que les positions paires, sauf au dernier passage. Exemple d'implantation (shel_tus) :

```
void tri_shell_tus(type_tus tab, int N)
 {
  int pt,dpg,P; /* position testée,dernier plus grand */
  composante tampon;
  for(P=1;P<=N/9;P=3*P+1); /* calcul de P initial (<=N/9) */
  for(;P>0;P/=3) /* inutile de soustraire 1 car division entière */
   {
    for(pt=P;pt<N;pt++)
     {
      dpg=pt−P;
      tampon=tab[pt];
      while(tab[dpg]>tampon&&dpg>=P−1)
                {tab[dpg+P]=tab[dpg];dpg−=P;}
      tab[dpg+P]=tampon;
     }
   }
 }
```

L'intérêt de ce tri, bien qu'il ait une boucle autour du tri par insertion, est qu'il crée rapidement un fichier presque trié, le dernier tri par insertion sera donc beaucoup plus rapide. Il est en général plus rapide que le tri par insertion pour les fichiers complètement mélangés, mais pour certains tableaux et pour certaines suites de P, il peut être bien plus mauvais que les autres tris.

4.1.4.6 le tri rapide (Quick Sort)

Ce tri est récursif. On cherche à trier une partie du tableau, délimitée par les indices *gauche* et *droite*. On choisit une valeur de ce sous−tableau (une valeur médiane serait idéale, mais sa recherche ralentit plus le tri que de prendre aléatoirement une valeur, par exemple la dernière), que l'on appelle *pivot*. Puis on cherche la position définitive de ce pivot, c'est à dire qu'on effectue des déplacements de valeurs de telle sorte que tous les éléments avant le pivot soient plus petits que lui, et que toutes celles après lui soient supérieures, mais sans chercher à les classer pour accélérer le processus. Puis on rappelle récursivement le tri de la partie avant

le pivot, et de celle après le pivot. On arrête la récursivité sur les parties à un seul élément, qui est donc nécessairement triée. (Quick_tus)

```
void tri_rapide_tus(type_tus tab,int gauche,int droite)
 {
  int g,d;
  composante tampon,val;
  if(droite<=gauche)return; /* fin de récursivité
                          si tableau d'une seule case à trier */
  /* choix du pivot : on prend par exemple la valeur de droite */
  val=tab[droite];
  g=gauche−1;
  d=droite;
  do
   {
    while(tab[++g]<val); /* g pointe le premier élément
              (à gauche) plus grand (ou égal) que le pivot */
    while(tab[−−d]>val); /* d pointe le premier élément
              (par la droite) plus petit (ou égal) que le pivot */
    if(g<d) /* si g et d ne se sont pas rencontrés, on
              échange les contenus de g et d et on recommence */
        {tampon=tab[g];tab[g]=tab[d];tab[d]=tampon;}
   }
  while(g<d); /* on sort quand g a rencontré d, alors tous les
      éléments à gauche de g sont <= au pivot, tous ceux
      à droite de d sont >= */
/* on place le pivot en position g (d serait aussi possible), donc dans sa
   bonne position (tous ceux à gauche sont <=, à droite sont >=) */
  tampon=tab[g];tab[g]=tab[droite];tab[droite]=tampon;
  /* il ne reste plus qu'à trier les deux parties, à droite
     et à gauche du pivot */
  tri_rapide_tus(tab,gauche,g−1);
  tri_rapide_tus(tab,g+1,droite);
 }
```

On appelle le tri d'un tableau complet par : `tri_rapide_tus(tableau, 0, N−1)`. On peut remarquer que cette implantation du tri n'est pas stable, mais peut l'être en gérant les égalités avec le pivot, mais en ralentissant le tri. On effectue dans la boucle deux tests (g<d), on peut supprimer le second par une boucle infinie et un break dans le if. Ce tri fait en moyenne 2Nlog(N) boucles, sur des fichiers bien mélangés, mais N^2 sur un fichier déjà trié (et dans ce cas la profondeur de récursivité est N, ce qui est prohibitif). Mais ces boucles sont longues et gourmandes en mémoire du fait de la récursivité: chaque appel de fonction est assez long, il faut mémoriser l'état actuel. On peut optimiser le tri en remplaçant la récursivité par des boucles (obligatoire si le langage utilisé n'est pas récursif), ce qui évite d'empiler des adresses, mais la gestion des variables locales doit être remplacée par gestion par pile (voir plus loin) pour mémoriser les sous−tris en attente, ce qui permettra d'accélérer le tri mais nécessite une programmation complexe (rappel : la récursivité sera automatiquement supprimée par le compilateur, cette transformation par le programmateur peut être plus efficace).

Plutôt que d'arrêter la récursivité sur des sous−tableaux de taille 1, on peut s'arrêter avant (entre 5 et 25 en général) pour éviter une profondeur de récursivité trop importante. Le fichier est alors presque trié, on peut alors effectuer un tri par insertion qui dans ce cas sera très rapide. Une autre amélioration possible est de mieux choisir le pivot. la solution idéale est de trouver à chaque fois la valeur médiane du sous−tableau à trier, mais sa recherche précise rend le tri plus lent que sans elle. Une solution quelquefois utilisée est de prendre par exemple trois valeurs, pour en prendre la valeur médiane, par exemple tab[droite], tab[gauche] et tab[(droite+gauche)/2] (dans le cas d'un fichier parfaitement mélangé, le choix de trois positions n'a pas d'importance, mais dans des fichiers presque triés le choix ci−dessus est plus judicieux). La totalité de ces améliorations peut apporter un gain de l'ordre de 20% par rapport à la version de base.

4.1.4.7 le tri par création

Lorsqu'il est nécessaire de disposer simultanément du tableau initial et du tableau trié, on peut recopier le tableau initial puis effectuer un tri sur la copie, ou adapter un des algorithmes précédents. Par exemple, à partir du tri par sélection, l'algorithme consiste à rechercher l'élément le plus petit, le copier en première position du tableau final, rechercher le suivant, le placer en seconde position, etc... En cas d'éléments identiques, il y a lieu de marquer les éléments déjà choisis, par exemple à l'aide d'un troisième tableau d'indicateurs (le tri est alors stable), ou suivant l'exemple (crea_tus) ci−dessous (ceci n'est qu'un exemple, d'autres possibilités existent) :

```
int le_suivant(type_tus ti, int taille, int precedent)
 {
  int pos,i;
/* 1) recherche du premier égal au précédent */
  pos=precedent+1;
  while(pos<taille&&ti[pos]!=ti[precedent])pos++;
  if(pos<taille)return(pos);
/* 2) sinon, recherche du suivant mais différent dans tout le tableau */
  pos=0;
  while(ti[pos]<=ti[precedent])pos++; /* le 1er > precedent */
  for(i=pos+1;i<taille;i++) /*le plus petit dans la suite */
    if(ti[i]>ti[precedent]&&ti[i]<ti[pos])pos=i;
  return(pos);
 }

void tri_creation_tus(type_tus ti, type_tus tf, int taille)
/* ti tableau initial, tf tableau final (trié) */
 {
  int i,imin=0;
  for(i=1;i<taille;i++)if(ti[i]<ti[imin])imin=i;
  tf[0]=ti[imin];
  for(i=1;i<taille;i++)
   {
    imin=le_suivant(ti,taille,imin);
    tf[i]=ti[imin];
   }
 }
```

On peut remarquer que ce tri minimise le nombre de copies des éléments, mais nécessite beaucoup de comparaisons de clefs (en particulier un élément déjà sélectionné sera encore comparé par la suite). Ceci peut être acceptable pour un fichier séquentiel à grands champs, sur bande par exemple, mais dont les clefs peuvent être stockées complètement en mémoire. On verra d'autres propositions dans le cas des fichiers.

4.1.4.8 d'autres tris

Suivant les données à trier, il peut être plus efficace de construire un algorithme de tri spécifique. Par exemple, si le tableau contient un grand nombre de valeurs similaires (exemple : gestion annuelle d'un stock où la plupart des articles entrent et sortent plusieurs fois par jour), on peut utiliser l'algorithme simple (par création) consistant à rechercher l'élément le plus petit, compter le nombre de ces éléments, les mettre dans le tableau destination, et répéter l'opération jusqu'à la fin du fichier destination. C'est le tri par comptage. Dans le cas où le nombre de clefs différentes est suffisamment faible, on peut utiliser un tableau de compteurs, ce qui permet d'effectuer le comptage en un seul balayage du fichier.

Dans la cas où les clefs sont bornées (c'est à dire comprises entre un minimum et un maximum connus à l'avance) et en nombre fini, on peut utiliser le tri basique : par exemple si toutes les clefs sont des entiers entre 000 et 999, on peut séparer le tableau en 10 parties en fonction des centaines, puis récursivement traiter les dizaines puis les unités (tri en base 10). Evidement, un tri en base 2 sera plus efficace sur ordinateur : on part à gauche ,on avance jusqu'à trouver un nombre commençant par 1, puis par la droite jusqu'à trouver un nombre commençant par 0, les échanger et continuer jusqu'à croisement des deux côtés. Puis on recommence

(récursivement par exemple) sur le bit suivant, jusqu'à tri complet. Pour trier des clefs alphabétiques, on peut effectuer un tri en base 26, sur les N premiers caractères (N pouvant valoir 2 ou 3 par exemple), le fichier est alors presque trié. Il est alors plus efficace d'effectuer un tri par insertion (passe de finition) plutôt que de répéter le tri basique jusqu'à tri complet.

Le tri par fusion utilise un algorithme de fusion de deux tableaux triés en un seul plus grand, appelé récursivement sur les deux moitiés du tableau, jusqu'à une taille de tableau de 1 (ou plus, avec un tri spécifique pour petits tableaux, par exemple par échange sur des sous−tableaux de 3 éléments)

4.1.5 recherches

On a souvent besoin de rechercher, dans un grand tableau, la position d'un élément donné. Un point particulier à ne pas oublier pour tous les algorithmes est le traitement du cas où l'élément cherché n'est pas dans le tableau. Une autre caractéristique importante d'un algorithme de recherche est son comportement désiré en cas d'éléments identiques (doit−il donner le premier, le dernier, tous ?).

4.1.5.1 la recherche séquentielle

Il suffit de lire le tableau progressivement du début vers la fin. Si le tableau n'est pas trié, arriver en fin du tableau signifie que l'élément n'existe pas, dans un tableau trié le premier élément trouvé supérieur à l'élément recherché permet d'arrêter la recherche, de plus cette position correspond à celle où il faudrait insérer l'élément cherché pour garder un tableau trié. Une recherche sur un tableau trié nécessitera en moyenne N/2 lectures, mais on se rapprochera de N pour un fichier non trié avec beaucoup de recherches d'éléments inexistants.

```
int rech_sequentielle_tus(type_tus tab, int N, composante val)
/* rend −1 si val non trouvée, première occurence trouvée sinon */
 {
  int i;
  for(i=0;i<N;i++)if(tab[i]==val)return(i);
  return(−1);
 }
```

4.1.5.2 la dichotomie

Dans le cas d'un tableau trié, on peut limiter le nombre de lectures à log(N)+1, en cherchant à limiter l'espace de recherche. On compare la valeur cherchée à l'élément central du tableau, si ce n'est pas la bonne, un test permet de trouver dans quelle moitié du tableau on trouvera la valeur. On continue récursivement jusqu'à un sous−tableau de taille 1. Il vaut bien mieux implanter cet algorithme de manière itérative, car la fonction se rappelle jusqu'à trouver la position désirée, puis seulement on effectue les dépilages, alors que l'on n'a plus besoin des états intermédiaires qui ont été mémorisés par la récursivité puisque le problème est résolu.

```
int rech_dichotomie_tus(type−tus tab, int N, composante val)
 {
  int g,m,d; /* gauche, milieu, droite */
  g=0;d=N;
  while (g<=d)
   {
    m=(g+d)/2; /* division entière */
    if(val<tab[m])d=m−1;
    else if(val>tab[m])g=m+1;
    else return(m)
   }
  return(−1);
 }
```

L'utilisation de la variable m permet d'éviter plusieurs calculs de (g+d)/2. Dans certains cas il peut être intéressant de prévoir également une mémorisation de tab[m], mais pas pour les tableaux puisqu'ils

permettent d'accès direct. L'ordre des tests est important, l'égalité ayant statistiquement moins de chances, elle doit être traitée en dernier (donc faire le maximum de tests dans le cas le moins probable). Si on avait traité l'égalité en premier, tous les autres cas auraient nécessité deux tests, l'égalité puis la sélection de la partie droite ou gauche. En cas de multiples éléments correspondants à la valeur cherchée, on retourne la position de l'un d'eux, mais pas nécessairement le premier ou le dernier. Pour retourner le premier par exemple, il suffit de rajouter une boucle testant l'égalité vers la gauche, à n'effectuer qu'une seule fois bien évidement, lorsque une valeur adéquate a été localisée. On peut améliorer l'algorithme en comparant val à tab[g] et tab[d], pour estimer la position recherchée plutôt que de la supposer au milieu, comme on effectue une recherche dans un dictionnaire : `m=g+(int)((val−tab[g])*(d−g)/(tab[d]−tab[g]))`. Attention, ce calcul se fait sur des composantes (par exemple des flottants), ce qui est toujours plus long que de simples calculs sur des entiers.

4.1.6 calculs mathématiques

Les tableaux unidimensionnels permettent de résoudre simplement divers problèmes mathématiques. Nous allons en traiter certains.

4.1.6.1 calcul vectoriel

L'utilisation des tableaux statiques est bien indiquée pour le calcul vectoriel. En effet, toutes les variables seront de même dimension. Il est aisé de prévoir une bibliothèque de fonctions vectorielles comportant toutes les opérations de base (multiplication par un réel, produit scalaire, produit vectoriel, produit mixte, norme...). Un grand nombre de problèmes géométriques se résoudront bien plus facilement par calcul vectoriel que de manière paramétrique.

4.1.6.2 polynômes

Une manière (mais il y en a d'autres) de représenter un polynôme est le tableau de ses coefficients. Par exemple $f(x)=4x^3+x+2$ sera représenté par le tableau 2,1,0,4 (en mettant le coefficient de x^i en position i). L'évaluation d'un polynôme (c'est à dire le calcul de sa valeur pour un x donné) ne doit pas utiliser de fonction puissances mais uniquement des multiplications successives, puisque toutes les puissances intermédiaires sont nécessaires. La somme de polynômes est triviale (somme des coefficients), le produit à peine plus compliqué (à moins que l'on ait besoin d'une optimisation poussée, dans ce cas on peut réduire d'1/4 le nombre de multiplications mais avec une complexité accrue). La résolution de l'équation f(x)=0 (recherche de racines) peut se faire par dichotomie par exemple (il faut alors donner le premier intervalle de recherche), en cas de racines multiples on en trouve une au hasard. Une recherche de racines plus efficace nécessite des algorithmes complexes, rarement universels (c'est à dire que dans certains cas ils ont un résultat déplorable, voire faux ou plantage de l'ordinateur en cas de divergence) qui ne seront pas traités ici, mais une littérature abondante existe sur ce sujet.

L'interpolation polynomiale correspond elle à la recherche d'une courbe polynomiale passant par N+1 points donnés : $P(x_i)=y_i$ pour i entre 0 et N. La solution la plus simple consiste à choisir le polynôme de Lagrange (d'ordre N):

$$P(x)=\sum_{j=0}^{N}\left(y_j \prod_{\substack{i=0 \\ i!=j}}^{N} (x-x_i)/(x_j-x_i)\right)$$

qui est le plus rapide à déterminer mais donne des résultats décevants pour N assez grand (100 par exemple), puisque les seules conditions imposées sont des points de passage, on obtient (souvent près des points

extrêmes) une courbe assez "folklorique" entre certains points. Une méthode souvent plus satisfaisante est l'utilisation des "splines", qui cherche parmi les multiples polynômes d'ordre N passant par N+1 points celui qui minimise une quadratique (en fait, correspond à la minimisation de l'énergie de déformation d'une poutre passant par ces points, d'où le nom de spline, "latte" en anglais) (voir mon support de cours sur l'infographie). Ou alors on peut utiliser une approximation (polynôme d'ordre M<N) passant au voisinage des points (par exemple moindres carrés).

4.2 tableaux multidimensionnels

Un tableau à N dimensions est en fait un tableau unidimensionnel de tableaux de N−1 dimensions. Tout ce qui a été présenté auparavant reste donc valable. La décision d'utiliser des tableaux multidimensionnels doit être bien réfléchie : ces tableaux nécessitant beaucoup de mémoire, il faut évaluer le ratio de composantes utiles par rapport aux places mémoire utilisées. Par exemple un tableau 10x10x10 utilisant dans chacune des 3 directions une seule fois les 10 mémoires prévues, les autres fois on s'arrête en moyenne à 7 (si ce n'est pas moins), réserve 1000 mémoires, dont seulement 352 utiles (7x7x7+3+3+3).

Outre les tableaux de chaînes de caractères, les tableaux multidimensionnels les plus utilisés sont les matrices Les algorithmes de base du calcul matriciel (base_mat) sont la mise à 0, la recopie, l'addition et le produit de matrices, qui sont simples à mettre en place (le produit entraîne néanmoins N^3 multiplications, des algorithmes plus performants existent mais ne commencent à être rentables que pour N très grand, entre 10000 et un million !).

Par contre, le problème de l'inversion d'une matrice est plus complexe. Elle est utilisée principalement pour la résolution de N équations à N inconnues, représentées par une matrice NxN. La méthode de Gauss est certainement la plus simple à mettre en oeuvre, bien que pouvant poser problème dans certains cas particuliers. On utilise en fait la méthode de résolution d'un système d'équations A.X=B par substitution. Nous allons le préciser sur un exemple :

1	1	−2		x_1		2
1	3	−4	*	x_2	=	6
−1	−2	6		x_3		−1
	A			**X**		**B**

On utilise le fait que de remplacer une ligne du système d'équations par une combinaison linéaire entre elle et d'autres lignes ne modifie pas le résultat, pour éliminer le premier élément de la seconde ligne. En soustrayant la première ligne à la seconde on obtient :

1	1	−2		x_1		2
0	2	−2	*	x_2	=	4
−1	−2	6		x_3		−1

Puis on élimine les deux premiers éléments de la troisième ligne (on ajoute la première puis on ajoute 1/2 fois la seconde) :

1	1	−2		x_1		2
0	2	−2	*	x_2	=	4
0	0	1		x_3		1

On peut désormais résoudre le système (en commençant par le bas) : $x_3=1$, donc (seconde ligne) $2x_2-2.1=4$, donc $x_2=3$, donc (première ligne) $x_1+3-2.1=2$ donc $x_1=1$.

```
void gauss_triangulation_simpliste(type_mat A,type_mat B, int N)
/* A de taille (N,N), B (N,1). On aurait pu gagner de la place en prenant B
   tableau unidimensionnel, ou même le rajouter en colonne N de A */
 {
  int ce,l,c;
  composante coef;
  for(ce=0;ce<N-1;ce++) /* ce=col à effacer (dans les lignes SUIVANTES) */
    for(l=ce+1;l<N;l++) /* pour chaque ligne au dessous */
     {
      coef=A[l][ce]/A[ce][ce];
      A[l][ce]=0; /* normalement pas besoin de le calculer */
      for(c=ce+1;c<N;c++) A[l][c]-=coef*A[ce][c];
      B[l][0]-=coef*B[ce][0];
     }
 }
```

Si ces substitutions font apparaître une ligne de 0, soit le système admet une infinité de solutions (0=0, une ligne est une combinaison linéaire des autres), soit aucune (0=N). Mais cette méthode n'est pas directement applicable pour des coefficients réels (ou du moins flottants). En effet, si les substitutions précédentes ont amené un coefficient nul, on obtient une division par zéro. S'il est proche de 0, l'utilisation de ce coefficient pour en annuler d'autres nécessitera un multiplicateur très grand, ce qui entraînera une multiplication importante de l'erreur inhérente à l'utilisation de réels et donc un résultat faux (en fait on se trouvera en présence de coefficients d'ordre de grandeur très différent, et donc les petits deviendront négligeables devant l'erreur sur les très grands). La solution est de ne pas traiter les lignes dans l'ordre mais pour une colonne donnée, choisir pour celle qui aura son premier terme non nul celle dont ce terme est le plus grand (on l'appelle le *pivot*). Il suffit alors d'échanger la ligne que l'on veut traiter avec la ligne contenant le pivot (échanger des lignes d'un système d'équations ne modifie pas la solution) (gauss):

```
void gauss_triangulation(type_mat A,type_mat B, int N)
 {
  int ce,l,c,lpivot;
  composante coef;
  for(ce=0;ce<N-1;ce++) /* ce=colonne à effacer */
   {
/*Recherche du pivot le + grand possible (dans les lignes qui restent)*/
    lpivot=ce;
    for(l=ce+1;l<N;l++)if(fabs(A[l][ce])>fabs(A[lpivot][ce]))lpivot=l;
/*Echange de la ligne du pivot et de la ligne ce (ne pas oublier B)*/
    for(c=ce;c<N;c++) /*tous les termes devant ce sont nuls*/
      {coef=A[ce][c];A[ce][c]=A[lpivot][c];A[lpivot][c]=coef;}
    coef=B[ce][0];B[ce][0]=B[lpivot][0];B[lpivot][0]=coef;
/*Suite de l'algorithme, comme le cas simpliste */
    for(l=ce+1;l<N;l++) /* pour chaque ligne au dessous */
     {
      coef=A[l][ce]/A[ce][ce];
      A[l][ce]=0;
      for(c=ce+1;c<N;c++)
        A[l][c]-=coef*A[ce][c];
      B[l][0]-=coef*B[ce][0];
     }
   }
 }
```

Une fois la matrice A triangulée, il ne reste plus qu'à déterminer la matrice colonne solution X:

```
void gauss_resolution(type_mat A,type_mat B,type_mat X,int N)
 {
  int l,c;
  composante tampon;
```

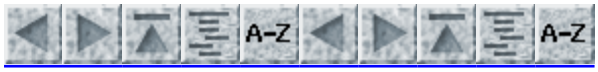
```
  for(l=N-1;l>=0;l--)
   {
    tampon=B[l][0];
    for(c=l+1;c<N;c++)tampon-=A[l][c]*X[c][0];
    X[l][0]=tampon/A[l][l];
   }
 }
```

Remarque : on pourrait retourner le résultat dans B (si l'utilisateur désire le garder, il lui suffirait de le copier avant). Ceci économise le tableau X et le tampon.

D'autres algorithmes existent, mais ils ne sont plus efficaces que Gauss que pour de très grosses matrices. Mais souvent les grosses matrices possèdent des propriétés particulières nécessitant d'utiliser d'autres structures de données que les tableaux à 2 dimensions (en cas de matrices triangulaires, symétriques, bandes, en "ligne de ciel", creuses..., voir paragraphe 11.3.2), pour lesquelles Gauss n'est pas la meilleure solution, car la triangulation supprime ces propriétés.

4.3 conclusions

Les tableaux unidimensionnels statiques sont d'une utilisation simple. Ils permettent un accès direct (donc quasi immédiat) à une donnée dont on connaît la position. Les seules manipulations de base rapides sont l'insertion et la suppression en fin du tableau. La dimension du tableau doit être connue (ou du moins maximisée) dès la phase d'écriture du programme, ces tableaux sont donc intéressants dans le cas de dimensions petites ou presque constantes.



5 LES TABLEAUX DYNAMIQUES

5.1 tableaux unidimensionnels en C

Les tableaux statiques nécessitent de maximiser lors de l'écriture du programme la dimension des tableaux. En cas de tableaux nombreux, il serait plus utile de ne réserver, pour chaque exécution, que la taille nécessaire à l'application en cours. En C, cette transformation est triviale grâce à la fonction malloc et à l'équivalence d'écriture entre les tableaux unidimensionnel et les pointeurs. Il suffira donc, pour utiliser les fonctions décrites pour les tableaux unidimensionnels statiques, de ne remplacer que la déclaration des tableaux :

```
#define composante float
typedef composante *type_tus; /* ou typedef composante type_tus[] */
```

reste identique, mais la déclaration `composante tab[dim]` sera remplacée par :

```
tab=(type_tus)malloc(taille*sizeof(composante));
```

à partir du moment où l'on a besoin du tableau, puis `free(tab);` quand il redevient inutile. Le problème des tableaux dynamiques en C est que l'on doit connaître la dimension d'un tableau avant sa première utilisation, ce qui empêchera par exemple une introduction de données du type "entrez vos données, tapez FIN pour terminer" mais nécessitera de demander en premier le nombre de données. L'autre problème provient du fait de la gestion de la mémoire inaccessible au programmeur : des créations et suppressions successives de tableaux vont créer une mémoire morcelée, et donc l'impossibilité de réserver un gros tableau, alors que la mémoire était disponible mais non continue. On est alors obligé de passer par la méthode du super−tableau pour pouvoir effectuer des retassages de mémoire.

5.2 la méthode du super−tableau

Dans les langages ne disposant pas de fonctions spécifiques à l'allocation dynamique de mémoire, on crée un grand tableau, appelé super−tableau, dimensionné au maximum de mémoire disponible. On crée ensuite les fonctions utilitaires nécessaires : réservation de mémoire (bloc) d'une taille donnée en argument (la fonction rend l'indice dans le super−tableau du début du sous−tableau alloué), libération d'un bloc préalablement alloué,... L'utilisation d'un bloc alloué est alors très simple : on remplace l'écriture `bloc[i]` par `super_tableau[indice_debut_bloc+i]`. Les parties du super−tableau utilisées n'ont pas absolument besoin d'une gestion poussée, mais on préfère en général prévoir un tableau annexe contenant les adresses et tailles des blocs actuellement réservés. Par contre il faut gérer les parties libres. Une solution est d'utiliser une pile (voir plus loin), mais la libération d'un bloc autre que le dernier soit nécessite un décalage des blocs suivants (attention, les indices de début de blocs sont modifiés), soit un marquage spécifique de cette zone (la place ne sera effectivement libérée que lorsque tous les blocs suivants seront libérés). Cette méthode est utilisable dans beaucoup de cas, il suffit de réserver en premier les blocs qui seront utilisés tout au long du programme, les blocs à durée de vie plus faible par la suite. L'autre solution consiste à gérer les blocs libres sous forme d'une liste chaînée (voir plus loin), chaque bloc libéré contenant l'indication de sa taille et l'adresse du bloc libre suivant (la place utilisée par ces informations est "masquée" puisque n'utilisant que des zones libres). L'allocation d'un bloc consiste alors en la recherche du premier bloc

libre de taille suffisante. Un retassage (et donc modification des indices de débuts de blocs) n'est nécessaire qu'en cas de mémoire trop morcelée.

Cette méthode du super−tableau permet de manière très simple d'accéder à toutes les possibilités des pointeurs, dans tout langage. C'est la raison pour laquelle des programmes très efficaces continuent à être développés dans ces langages (pratiquement tous les gros programmes scientifiques (C.A.O., éléments finis,...) de plusieurs centaines de milliers de lignes sont écrits en FORTRAN, et utilisent cette méthode). De plus la gestion par le programmeur de l'allocation de mémoire peut permettre d'être plus efficace que la gestion par le compilateur, puisque pouvant être spécifique au problème traité.

5.3 les tableaux multidimensionnels

Il n'y a pas en C d'équivalence d'écriture permettant une utilisation directe de l'allocation dynamique pour les tableaux multidimensionnels. On utilise donc la même méthode pour l'allocation dynamique en C (la mémoire est en fait un tableau unidimensionnel d'octets) qu'avec un super−tableau. Nous n'allons traiter que les tableaux à deux dimensions, l'extension à N dimensions ne posant aucun problème.

5.3.1 matrices pleines (matrices rectangulaires dynamiques : mrd)

Pour allouer dynamiquement une matrice de taille (nbl,nbc), on réserve un tableau unidimensionnel de taille nbl*nbc :

```
typedef composante * mat_dyn;
mat_dyn alloc_mrd(int nbl;int nbc)
 {return((mat_dyn)malloc(nbl*nbc*sizeof(composante)));}
```

On accède à l'élément en ligne l et colonne c par m[l*nbc+c] (pour l et c commençant à 0, comme en C). On peut prévoir une fonction (ou même une macro, plus rapide) pour effectuer ce calcul :

```
#define adr_mrd(l,c,nbc) ((l)*(nbc)+(c))
```

Les algorithmes pour les matrices statiques doivent être réécrits, en remplaçant chaque mat[l][c] par mat[adr_mrd(l,c,nbc)]. Souvent, on choisira un nom plus court pour adr_mrd. Dans les cas simples uniquement, on peut choisir nbc comme variable globale pour éviter sa transmission en argument (par exemple si l'on utilise des matrices carrées toutes de même taille), mais ce n'est pas conseillé (dans le cas d'une macro, ce ne sont pas de vrais passages d'arguments et donc ne prend pas de temps supplémentaire).

Exercice (gauss_mrd) : écrire une version de la méthode de Gauss pour des matrices dynamiques. La solution donnée en annexe 2 (du document papier) est un exemple de matrices dynamiques.

5.3.2 tableaux de tableaux dynamiques

Dans le cas de matrices non pleines, la gestion sera plus complexe mais le résultat très efficace. On gère en fait un ensemble de lignes de longueur différente (donc des tableaux dynamiques). Dans quelques cas, la longueur et la position de chaque ligne se trouve par calcul : matrices triangulées, matrices bandes (réorganisées pour que tous les éléments non nuls soient au maximum à la distance B (largeur de bande) de la diagonale),... Un utilise alors la même méthode que pour les matrices pleines, en changeant simplement le calcul pour accéder à une composante (adr_mrd). Dans tous les autres cas, chaque ligne est allouée dynamiquement, l'adresse et la longueur de chaque ligne étant stockée dans un tableau statique ou dynamique, voire liste chaînée... La longueur peut être omise si la ligne est terminée par un signe particulier (chaînes de caractères en particulier). Les manipulations deviennent alors plus efficaces : les manipulations de composantes (dans les lignes) restent des manipulations de type tableaux, donc efficaces tant que les lignes ne sont pas de taille exagérée, alors que les manipulations de lignes (plus volumineuses) se font par manipulation d'adresses (par exemple pour échanger deux lignes, il suffit d'échanger les deux adresses de

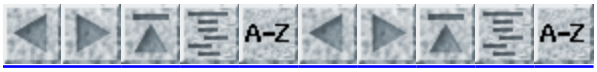
lignes, les lignes elles−mêmes n'étant physiquement pas déplacées). Ce type de données est certainement le plus efficace pour du traitement de textes. Mais il permet aussi de traiter les matrices en "ligne de ciel:" dans le cas (fréquent en mécanique) de matrices symétriques contenant beaucoup de 0, on peut avoir du mal à réorganiser la matrice efficacement pour qu'elle soit sous forme de bande, par contre il est plus facile de la réorganiser pour qu'un maximum de lignes soient regroupées près de la diagonale, quelques lignes restant à grande largeur. On stocke alors uniquement, pour chaque ligne, les éléments de la diagonale au dernier non nul. Ce type de stockage est également appelé quelquefois "à largeur de bande variable".

Ces tableaux de tableaux dynamiques permettent toujours un accès presque direct à un élément l,c : tab[adresse_ligne[l]][c]. Mais ils gardent les problèmes des tableaux dynamiques, en particulier de ne pas être totalement dynamiques puisqu'une fois une ligne créée, on ne peut pas directement l'agrandir (en fait il suffit de réserver une nouvelle ligne plus grande, y recopier la précédente, changer l'adresse de la ligne dans le tableau d'adresses de lignes puis supprimer l'ancienne ligne). Souvent dans ce cas on préfère directement créer un ligne un peu plus longue que nécessaire, pour ne pas répéter l'opération trop souvent.

Exercice (inse_ttd) : écrire un programme pour trier des lignes de texte, utilisant les tableaux de tableaux dynamiques. On choisira un tri par insertion, puisque les échanges de lignes ne sont que des échanges d'adresses, le texte en lui−même ne sera jamais déplacé.

5.4 conclusions

Les tableaux dynamiques ne sont pas totalement dynamiques, c'est à dire que leur taille, bien que définie en cours d'exécution du programme, ne peut pas facilement être modifiée (c'est néanmoins presque toujours possible mais relativement coûteux en temps et mémoire). Mais ils gardent la principale qualité des tableaux statiques : l'accès immédiat à une composante dont on connaît la position. Ils en gardent également la faible efficacité en cas d'insertions et suppressions. Les tableaux dynamiques sont néanmoins la solution minimisant la place occupée en mémoire pour de gros ensembles de données : Seule la place nécessaire est réservée (excepté une mémoire pour l'adresse du début du tableau), aucune mémoire n'est utilisée pour indiquer où se trouve la composante suivante, puisqu'elle est placée directement après la précédente (contrairement aux listes par exemple).



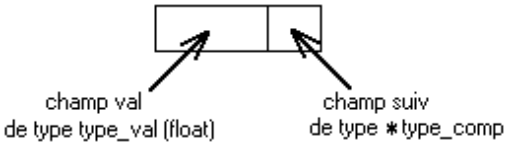
6 LES LISTES

6.1 fonctions de base et manipulations (base_lst)

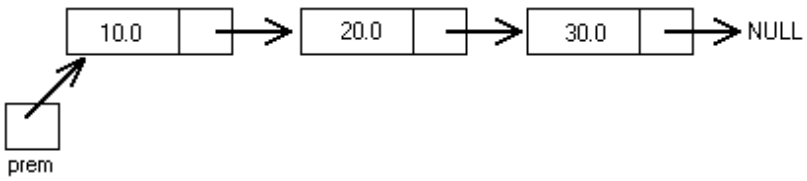
Les listes sont un regroupement ordonné de données effectué de manière à ce que chaque composante sache où se trouve la suivante. En C, une composante sera une structure contenant la valeur mémorisée, mais également un pointeur sur la composante suivante. Sur fichier ou dans les langages ne possédant pas de pointeurs, on utilisera la méthode du super−tableau, le pointeur étant remplacé par l'indice, dans le super−tableau, de la composante suivante. Une liste est accessible par l'adresse de sa première composante. On supposera dans la suite que les valeurs à mémoriser sont de type flottant, mais évidement tout autre type de données est possible (même tableaux, structures, listes...). La déclaration en C sera donc :

```
typedef float type_val;
typedef struct scomp {type_val val; struct scomp *suiv;} type_comp;
typedef type_comp *adr_comp;
```

Nous représenterons les composantes des listes (de type type_comp) ainsi :



Supposons disposer de la liste ci−dessous en mémoire (nous verrons plus loin comment la créer en mémoire). La variable `prem`, de type `adr_comp` (donc pointeur sur une composante) contient l'adresse de la première composante. Le champ `suiv` de la dernière composante contient la valeur NULL, donc ne pointe sur rien.



Pour simplifier les algorithmes, on peut mettre d'office une composante particulière en début et fin de liste (appelée *sentinelle*). Ceci évite de devoir traiter de manière particulière le premier et dernier élément de la liste, puisque chaque composante utile possède toujours un précédent et un suivant. Cette méthode prend un peu plus de mémoire (négligeable en % pour de longues listes) et évite des tests systématiques dans le boucles, alors qu'ils ne servent que pour les extrémités (d'où gain de temps appréciable, toujours en cas de longues listes). Une autre méthode pour repérer le dernier élément est de le faire pointer sur lui−même. Nous utiliserons dans la suite des listes sans sentinelle, le dernier élément pointant sur NULL.

L'appel de : `affiche_lst(prem)` affichera à l'écran le contenu de la liste :

```
void affiche_lst(adr_comp l)
 {
  while(l!=NULL)
   {
    printf("%6.1f ",l->val);
    l=l->suiv;
   }
  printf("\n");
 }
```

l pointe sur la première composante. On affiche la valeur pointée par l puis on fait pointer l sur son suivant. On répète le processus jusqu'en fin de liste (lorsque l'on pointe sur NULL). La variable l doit être locale (donc passée par valeur) pour ne pas modifier le contenu de prem, et donc perdre l'adresse du début de la liste, ce qui empêcherait tout accès ultérieur à la liste.

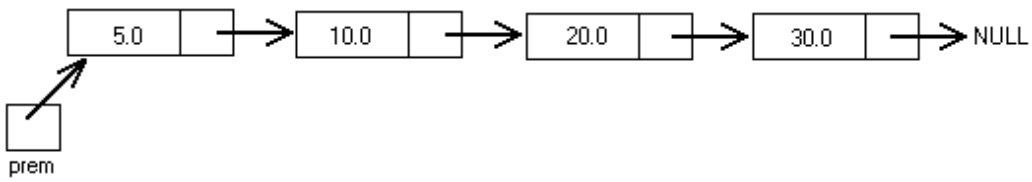
Comme pour les chaînes de caractères, plutôt que de gérer un variable entière indiquant toujours la longueur actuelle de la liste, c'est la spécificité du dernier élément (ici, le champ suiv contient NULL) qui permet de préciser que l'on est en fin de liste. Pour connaître la longueur d'une liste, on utilisera :

```
int longueur_lst(adr_comp l)
 {
  int n=0;
  while(l!=NULL)
   {
    l=l->suiv;
    n++;
   }
  return(n);
 }
```

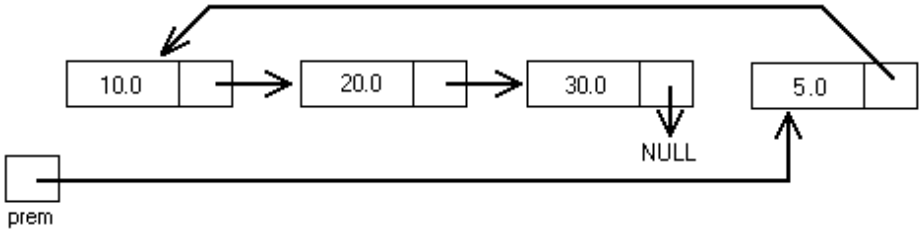
Les listes ont l'avantage d'être réellement dynamiques, c'est à dire que l'on peut à loisir les rallonger ou les raccourcir, avec pour seule limite la mémoire disponible (ou la taille du super−tableau). Par exemple pour insérer une nouvelle composante en début de liste, on utilisera :

```
void insert_premier(adr_comp *prem,type_val val)
 {
  adr_comp nouv;
  nouv=(adr_comp)malloc(sizeof(type_comp));
  nouv->val=val;
  nouv->suiv=*prem;
  *prem=nouv;
 }
```

En insérant la valeur 5 à notre exemple précédent, on obtient la liste :

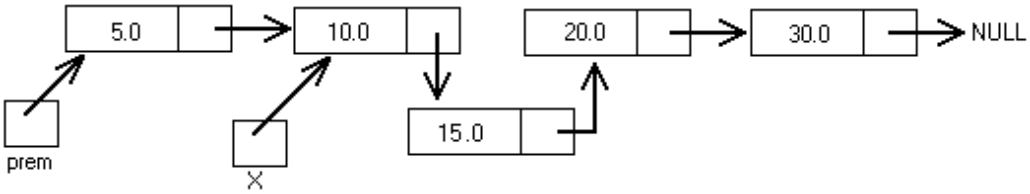


La variable `prem` a dû être passée par adresse, pour que l'on récupère bien l'adresse de la nouvelle première composante (appel : `insert_premier(&prem,5)`). En fait le schéma ci−dessus n'est qu'une représentation abstraite de la liste (chaque composante pointe sur la suivante), alors que les composantes sont physiquement placées différemment en mémoire. Dans le meilleur des cas, la nouvelle composante créée a été placée juste derrière celles déjà existantes (premier emplacement libre). Un schéma plus proche de la réalité serait donc :



Ces deux schémas sont équivalents (les liens partent des mêmes cases, et pointent sur les mêmes cases), seule la disposition des cases diffère. En fait ceci nous montre bien que la disposition réelle en mémoire ne nous intéresse pas (jamais la valeur effective de l'adresse contenue dans un pointeur ne nous intéressera). Dans nos schémas, nous choisirons donc une disposition des composantes correspondant plus au problème traité qu'à la disposition réelle en mémoire (par exemple dans le schéma suivant, on utilise une représentation bidimensionnelle, alors que la mémoire n'est qu'unidimensionnelle).

Le second avantage des listes est que l'insertion d'une composante au milieu d'une liste ne nécessite que la modification des liens avec l'élément précédent et le suivant, le temps nécessaire sera donc indépendant de la longueur de la liste. Supposons disposer d'une variable X de type `adr_comp`, pointant sur la composante de valeur 10. L'appel de `insert_après(X,15)` donnera le résultat suivant :




```
void insert_après(adr_comp prec,type_val val)
 {
  adr_comp nouv;
  nouv=(adr_comp)malloc(sizeof(type_comp));
  nouv->val=val;
  nouv->suiv=prec->suiv;
  prec->suiv=nouv;
 }
```

Cette fonction permet l'insertion en tout endroit de la liste, sauf en première position (qui n'a pas de précédent dans notre cas puisque sans sentinelle), il faut traiter spécifiquement le premier de la liste (voir plus haut). L'insertion nécessite la connaissance de l'adresse du précédent et du suivant. Les composantes ne comportant que des liens vers l'élément suivant, il faut nécessairement donner à la fonction insert_après l'adresse du précédent, alors que d'habitude on préfère donner l'élément devant lequel on veut faire l'insertion (tableaux par exemple). Ceci nous montre le principal problème des listes : l'accès est séquentiel. On devra, pour chercher la composante précédente, parcourir toute la liste depuis son début (on donne l'adresse du début de la liste et de l'élément dont on cherche le précédent):

```
adr_comp rech_prec(adr_comp prem, adr_comp suiv)
/* rend l'adresse, NULL si non trouvé */
 {
  while(prem->suiv!=suiv && prem!=NULL)
     prem=prem->suiv;
  return(prem);
 }
```

Une autre solution, si l'on a fréquemment besoin de parcourir la liste vers l'avant et vers l'arrière, est de stocker dans chaque composante l'adresse du suivant et celle du précédent. Ceci nécessite plus de place en mémoire et ralentit un peu les manipulations de base (il y a plus de liens à mettre à jour). Mais même dans ce cas, l'accès reste séquentiel. Ceci empêche d'utiliser un certain nombre d'algorithmes utilisant l'accès direct (la dichotomie par exemple). Pour trouver l'adresse du Nième élément d'une liste il faudra utiliser (on suppose numéroter 0 le premier) :

```
adr_comp rech_ind(adr_comp l, int i)
/* rend l'adresse du (i+1)ième, NULL si liste trop courte */
 {
  int j;
  for(j=0;j<i && l!=NULL;j++) l=l->suiv;
  return(l);
 }
```

La recherche d'un élément, même si la liste est triée, se fera donc toujours séquentiellement :

```
adr_comp rech_val(adr_comp l, type_val v)
/* rend l'adresse, NULL si non trouvé */
 {
  while(l!=NULL && l->val!=v) l=l->suiv;
  return(l);
 }
```

Pour créer une liste, la solution la plus simple consiste à boucler sur un appel de la fonction `insert_premier`, mais les éléments seront stockés dans l'ordre inverse de leur introduction (le dernier saisi sera placé en premier). Pour une création de liste dans l'ordre, on fera :

```
adr_comp saisie_lst(void)
 {
  adr_comp prem=NULL,prec,actu; /* premier, précédent, actuel*/
  type_val v;
  int err;
  do
   {
    printf("entrez votre prochaine valeur, un caractère pour finir :");
    err=scanf("%f",&v);
    if(err<=0)break; /*scanf rend le nombre de variables lues sans erreur*/
    actu=(adr_comp)malloc(sizeof(type_comp));
    actu->val=v;
    if(prem==NULL)prem=actu; else prec->suiv=actu;
    prec=actu;
   }
  while(1);
  actu->suiv=NULL;
  return(prem);
 }
```

Cette fonction crée la liste en mémoire, effectue la saisie et retourne l'adresse du début de la liste.

Il nous reste à traiter les suppressions dans une liste. Il faut ici encore préciser l'élément précédent celui à supprimer, et traiter de manière particulière le début de la liste :

```
void suppr_suivant(adr_comp prec)
 {
  adr_comp a_virer;
  if(prec==NULL || prec->suiv==NULL)
    {puts("rien à supprimer");return;}
  a_virer=prec->suiv;
  prec->suiv=a_virer->suiv;
  free(a_virer);
 }

void suppr_premier(adr_comp *prem)
 {
  adr_comp a_virer;
  if(*prem==NULL) {puts("rien à supprimer");return;}
  a_virer=*prem;
  *prem=(*prem)->suiv;
  free(a_virer);
```

```
 }
```

La suppression complète d'une liste permet de récupérer la place en mémoire. Cette opération n'est pas nécessaire en fin de programme, le fait de quitter un programme remet la mémoire dans son état initial et libère donc automatiquement toutes les mémoires allouées dynamiquement :

```
void suppr_tout(adr_comp prem)
/* attention : ne met pas NULL dans prem qui pointe donc toujours sur la
liste, qui n'est plus allouée mais dont le contenu n'a peut−être pas
changé */

 {
  adr_comp deuxieme;
  while(prem!=NULL)
   {
    deuxieme=prem−>suiv;
    free(prem);
    prem=deuxieme;
   }
 }
```

Il n'était à priori pas obligatoire d'utiliser la variable deuxième car la libération par **free** n'efface pas les mémoires, le suivant est donc encore disponible après free et avant tout nouveau malloc. Néanmoins cette écriture est plus sure, ne gérant pas la mémoire nous même (sauf si méthode du super−tableau), en particulier en cas de multitâche, d'interruption matérielle...

On trouvera un exemple utilisant ces fonctions dans test_lst.c (disquette d'accompagnement)

6.2 les tris

Seuls les tris n'utilisant pas l'accès direct pourront être efficaces pour les listes. Au lieu de déplacer les valeurs dans la liste, on changera uniquement les liens. Dans la plupart des configurations, le tri par insertion sera le plus efficace.

6.2.1 le tri bulle

Le tri bulle est donc assez efficace (n'ayant pas de lien sur la composante précédente, on mémorisera toujours l'adresse du précédent dans une variable auxiliaire). Mais il reste peu efficace lorsque des éléments sont loin de leur position finale, puisque chaque passage ne peut déplacer un élément que d'une position (de l'ordre de $N^2/2$ échanges, autant de boucles internes), on le réservera donc aux listes presque triées. Les autres tris, comme le tri par insertion déplaceront chaque élément directement en bonne position, mais le temps nécessaire à la recherche peut être assez long, alors qu'ici la position destination est directement connue (mais pas nécessairement juste) (bull_lst) :

```
void tri_bulle_lst(adr_comp *prem)
/* le premier ne sera peut−être plus le même donc passage par adresse */
 {
  int ok;
  adr_comp prec,actu,suiv;
  do
   {
    ok=1; /* vrai */
    prec=NULL;
    actu=*prem;
    suiv=actu−>suiv;
    while(suiv!=NULL)
     {
      if(actu−>val > suiv−>val)
       {
```

```
        ok=0;
        if(prec!=NULL) prec->suiv=suiv; else *prem=suiv;
        actu->suiv=suiv->suiv;
        suiv->suiv=actu;
       }
      prec=actu;
      actu=suiv;
      suiv=actu->suiv;
     }
   }
  while(!ok);
 }
```

En utilisant `prec->suiv` et `prec->suiv->suiv`, on pouvait éviter d'utiliser les variables `actu` et `suiv`. Le temps d'accès aux variables aurait été un peu plus long mais on supprimait deux des trois affectations situées en fin de boucle.

6.2.2 le tri par insertion

Le tri par insertion sera bien plus intéressant, dans la majorité des cas : il nécessite une boucle principale (séquentielle), dans laquelle on appelle une seule recherche pour placer l'élément, les déplacements se faisant très rapidement et sans s'occuper du reste de la liste (inse_lst) :

```
void tri_insertion_lst(adr_comp *prem)
 {
  /*position testée, précédent,dernier plus petit*/
  adr_comp pt,prec,dpp;

  for(prec=*prem,pt=(*prem)->suiv;pt!=NULL;prec=pt,pt=pt->suiv)
   if(prec->val>pt->val) /*inutile de chercher si en bonne
position */
   {
    prec->suiv=pt->suiv;
    if((*prem)->val > pt->val) /*cas particulier du premier*/
     {
      pt->suiv=*prem;
      *prem=pt;
     }
    else
     {
      dpp=*prem;
      while(dpp->suiv->val <= pt->val)dpp=dpp->suiv;
 /* on est sur d'en trouver un, vu les tests effectués plus haut */
      pt->suiv=dpp->suiv;
      dpp->suiv=pt;
     }
   }
 }
```

Ici également, on aurait pu tenter d'éviter l'utilisation des deux variables prec et pt puisque pt est le suivant de prec, mais les échanges auraient alors nécessité une variable tampon. Cette version du tri est stable (le nouveau est inséré derrière les valeurs égales), mais ceci ralentit un peu la recherche de la position définitive d'une valeur en cas de nombreuses valeurs égales (si la stabilité n'est pas nécessaire, il suffit d'arrêter la recherche au premier élément supérieur ou égal). Dans tous les cas, on fera au maximum N échanges (aucun en bonne position), en moyenne N/2 pour les fichiers mélangés. Dans la cas d'un fichier mélangé, le nombre de boucles internes est de N(N−1)/2 en moyenne. Dans le cas d'un fichier presque trié, dans le cas de quelques éléments (en nombre P) pas du tout à leur place, ce tri est très efficace : P échanges, (P+2)*N/2 boucles internes, donc on devient linéaire en N. Par contre dans le cas de nombreux éléments pas tout à fait à leur place (à la distance D), il l'est moins que le tri bulle, la recherche de la position exacte se faisant à partir du début de la liste (proche de N^2 : (N−D)(N−1) boucles internes). Dans ce cas, si l'on peut également disposer d'un chaînage arrière, on fera la recherche à partir de la position actuelle, ce qui rendra le tri très

efficace également dans ce cas : D(N−1) boucles internes donc linéaire en N. Sinon, on peut mémoriser la position du Dième précédent, le comparer à la valeur à insérer et donc dans la majorité des cas (si D bien choisi) rechercher derrière lui, dans les quelques cas où il faut l'insérer devant lui, on effectue une recherche depuis le début.

6.2.3 le tri par sélection

Le tri par sélection prendra environ $N^2/2$ boucles internes, et ceci quel que soit l'ordre initial du fichier. Le nombre maximal d'échanges sera de N, N/2 en moyenne pour un fichier mélangé, P pour uniquement P éléments mal placés (sele_lst) :

```
void tri_selection_lst(adr_comp *prem)
 {
  type_comp faux_debut;
  adr_comp pdpd,pdpp,i;
  faux_debut.suiv=*prem;
  for(pdpd=&faux_debut;pdpd->suiv!=NULL;pdpd=pdpd->suiv)
   {
    /*recherche du plus petit (du moins son précédent)*/
    pdpp=pdpd; /*le plus petit est pour l'instant l'actuel, on va tester les suivants*/
    for(i=pdpd->suiv;i->suiv!=NULL;i=i->suiv)
           if(i->suiv->val < pdpp->suiv->val)pdpp=i;
    /* échange (si beaucoup d'éléments
déjà en place, rajouter un test pour ne pas échanger
inutilement) */
    i=pdpp->suiv;
    pdpp->suiv=i->suiv;
    i->suiv=pdpd->suiv;
    pdpd->suiv=i;
   }
  *prem=faux_debut.suiv; /* retourner le bon premier */
 }
```

pdpd représente le Précédent De la Place Définitive (boucle principale : de place définitive valant *prem jusqu'à fin de la liste), pdpp représente le Précédent Du Plus Petit (de la suite de la liste, pas encore traitée). On a dans cette implantation évité les variables pour la place définitive et le plus petit, ceux−ci étant les suivants des variables précisées ci−dessus. Ceci nuit à la clarté mais améliore l'efficacité. On pouvait améliorer la clarté (définir les "variables" pd et pp) sans nuire à l'efficacité (ne pas devoir les remplacer par leur suivant en fin de boucle) par des #define :

```
#define pd pdpd->suiv
#define pp pdpp->suiv
```

Pour limiter la portée de ces substitutions à cette fonction, il suffit de déclarer juste derrière la fin de la fonction :

```
#undef pd
#undef pp
```

De plus, afin d'éviter de tester partout le cas particulier du premier de la liste, on a choisi d'ajouter un élément en début de liste (faux_début), pointant sur le premier de la liste. Dans ce cas, tous les éléments y compris le premier ont un précédent. Le gain est donc appréciable (en temps car moins de tests et en taille du code source), pour un coût limité à une variable locale supplémentaire.

6.2.4 le tri par création

Ce tri nécessite de recréer tous les liens. En fait on utilisera un algorithme par insertion (ou par sélection) un peu modifié, puisque ces deux tris créent progressivement la liste triée. en cas de valeurs de grande taille, on

préférera ne créer qu'une nouvelle liste de liens, pour éviter de doubler la place mémoire utilisée par les valeurs (la composante de base contiendra donc deux pointeurs : l'adresse de la valeur et l'adresse de la composante suivante).

6.2.5 les autres tris

Le tri shell, quand à lui, n'améliore pas le tri par sélection puisqu'il traite les composantes séparées d'une distance P.

Pour le tri rapide, le nombre de boucles internes est de l'ordre de Nlog(N) (N sur toute la liste, puis deux fois N/2 sur les deux moitiés.... donc N*P, P étant la profondeur de récursivité, qui est de log(N)). Par contre les échanges sont plus nombreux que les autres tris (en moyenne une boucle interne sur deux pour un fichier mélangé). Pour transposer l'implantation détaillée pour les tableaux, il faut disposer d'une liste à chaînage avant et arrière, mais il est facile de n'utiliser que le chaînage avant : l'algorithme devient donc :

- choix d'un pivot (le premier par exemple, puisqu'on y accède directement),
- scruter la liste pour créer deux sous−listes (celle des valeurs plus petites et celle des plus grandes),
- appel récursif sur les deux sous−listes.
- mise à jour des liens (fin de la première liste, pivot, début de la deuxième)

Cette gestion des liens ralentit beaucoup l'algorithme, son implantation doit donc être soigneusement optimisée pour espérer un gain, du moins pour les très longues listes.

6.3 problèmes mathématiques

On utilisera les listes pour les cas nécessitant de nombreuses manipulations (en particulier dans les problèmes graphiques). On les utilise également pour les polynômes ou matrices creuses (c'est à dire avec de nombreux coefficients nuls, on ne stocke que les coefficients non nuls (ainsi que leur indice ou le nombre de 0 qui les séparent du suivant). Les algorithmes restent similaires à ceux s'appliquant aux tableaux, mais sont souvent moins efficaces (une triangulation de Gauss rend non nuls une grande partie des coefficients nuls, nécessitant de nombreuses insertions de nouveaux éléments).

6.4 conclusions

Les listes sont parfaitement dynamiques. Toutes les modifications peuvent se faire en cours d'utilisation. Par contre l'accès est séquentiel, ce qui peut être très pénalisant dans certains cas. Il est néanmoins possible d'utiliser un double chaînage avant et arrière, mais au détriment de la place mémoire. L'encombrement des listes en mémoire est important : chaque élément doit contenir l'adresse du suivant (alors que pour les tableaux elle était facile à calculer donc non stockée). Pour de très grandes listes, on peut remédier à ce problème en utilisant des listes de tableaux, mais l'utilisation devient complexe. Un autre avantage des listes est que l'on utilise toujours un adressage indirect, et donc que les manipulations dans les listes ne font que des modifications du chaînage, sans réellement déplacer les valeurs stockées, ce qui est capital en cas de champs de grande taille.

7 LES PILES ET FILES

7.1 définition

Ce sont des structures de données ordonnées, mais qui ne permettent l'accès qu'à une seule donnée. On utilise souvent le nom générique de *pile* pour les piles et les files, un seul nom existant en anglais (stack). Les piles (stack LIFO : Last In First Out) correspondent à une pile d'assiettes : on prend toujours l'élément supérieur, le dernier empilé. Les files (on dit aussi *queues*) (stack FIFO: First In First Out) correspondent aux files d'attente : on prend toujours le premier élément, donc le plus ancien (on ne tolère pas ici les resquilleurs). Les piles et files sont très souvent utiles : elles servent à mémoriser des choses en attente de traitement. Elles permettront une clarification des algorithmes quand effectivement on n'a pas besoin d'accéder directement à tous les éléments. Elles sont souvent associées à des algorithmes récursifs. Il n'y a pas de structures spécifiques prévues dans les langages (sauf FORTH), il faut donc les créer de toutes pièces. Pour les piles, on utilisera un tableau unidimensionnel (statique ou dynamique) en cas de piles de hauteur maximale prévisible (la hauteur de la pile est mémorisée par une variable entière), ou une liste en cas de longueur très variable (ne pas oublier que dans ce cas on a un surcoût en mémoire d'autant de liens (pointeurs) que d'éléments empilés). Pour les files, l'utilisation d'un tableau nécessite deux variables : la position du premier et celle du dernier. La suppression du premier élément ne se fait pas par décalage des suivants mais en incrémentant la variable indiquant le premier. La gestion est alors un peu plus complexe que pour les piles, puisque le suivant de la fin du tableau est le début du tableau (en fait, l'indice du suivant est l'indice plus 1, modulo la taille du tableau. La fonction modulo est en fait très rapide pour les nombres correspondants à une puissance de 2, à condition de l'implanter à l'aide d'un masquage. L'utilisation d'une liste pour une file par contre est aussi simple que pour une pile.

7.2 fonctions de base

Les fonctions de base pour les piles sont l'empilage et le dépilage, pour les files l'enfilage et le défilage. Dans les deux cas on prévoira également un fonction d'initialisation, et une fonction indiquant si la pile est vide. Seules ces fonctions de base dépendent de la méthode réelle de mise en oeuvre (tableau, liste,...). Tous les algorithmes n'utiliseront que ces fonctions. C'est le grand avantage des piles et files, puisque l'on va pouvoir modifier facilement le type d'implantation en mémoire sans réécrire les programme. Ceci permet de tester d'abord la faisabilité du programme sur des petites quantités de données puis seulement on l'optimise pour l'utilisation réelle. Pour une implantation par tableaux, on écrira par exemple, pour une pile de flottants (base_p_t) :

```
#define dim_pile 100
#define composante float
/* static pour empêcher l'accès direct extérieur*/
static composante pile[dim_pile];
static int sommet;

void init_pile(void)
{sommet=0;}

int pile_vide(void)
{return(sommet=0);}

int empiler(composante x)
/* retourne 0 si pas d'erreur (donc il restait de la place dans la pile) */
{
```

```
 if(sommet<dim_pile)
   {pile[sommet++]=x;return(0);}
 else
   {puts("pile saturée");return(1);}
}

composante depiler(void)
{
 if (sommet>0) return(pile[−−sommet]);
 else puts("pile vide");return(0);
}
```

Si l'on désire un dimensionnement totalement dynamique de la pile, on utilisera une liste (base_p_l) :

```
#include <alloc.h>
#include <stdio.h>
#define composante float
typedef struct s_comp
   {composante val;struct s_comp *prec;}type_comp;

static type_comp *sommet=NULL;

int empiler(composante x)
/* retourne 0 si pas d'erreur (donc il restait de la place dans la pile)*/
{
 type_comp *nouv;
 if((nouv=(type_comp*)malloc(sizeof(type_comp)))!=NULL)
  {
   nouv−>val=x;
   nouv−>prec=sommet;
   sommet=nouv;
   return(0);}
 else
   {puts("pile saturée");return(1);}
}

composante depiler(void)
{
 composante x;
 if (sommet!=NULL)
  {
   x=sommet−>val;
   free(sommet);  /*pour plus de sûreté, on peut passer par
                       une variable*/
   sommet=sommet−>prec;
   return(x);
  }
 else puts("pile vide");return(0);
}

void init_pile(void)
{while(sommet!=NULL)depiler();}

int pile_vide(void)
{return(sommet==NULL);}
```

Pour certaines applications, on pourra préférer ne pas libérer la mémoire au dépilage pour gagner du temps : au dépilage, mais aussi à l'empilage, on ne prend du temps que s'il faut agrandir la pile. On peut également utiliser une pile de tableaux dynamiques, ce qui permet d'économiser de la place mémoire, sans être limité en taille.

Pour les files, l'implantation des fonctions de base est similaire, par exemple par tableaux (base_f_t) :

```
#define dim_file 100
#define composante float
static composante file[dim_file];
static int bas,sommet,taille;
#define suiv(i) ((i)<dim_file−1 ? ((i)+1) : 0)

void init_file(void)
{bas=sommet=taille=0;}

int file_vide(void)
{return(taille=0);}

int enfiler(composante x)
/* retourne 0 si pas d'erreur (donc il restait de la place dans la pile)
*/
{
 if(taille<dim_file)
   {file[sommet]=x;sommet=suiv(sommet);taille++;return(0);}
 else
   {puts("file saturée");return(1);}
}

composante depiler(void)
{
 composante x;
 if (taille>0) {x=file[bas];bas=suiv(bas);taille−−;return(x);}
 else {puts("file vide");return(0);}
}
```

Il faut prévoir un indice pour chaque extrémité de la file. Il en serait de même pour une implantation par liste chaînée (un pointeur pour chaque extrémité).

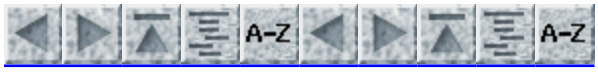
7.3 utilisations

Les piles sont souvent nécessaires pour rendre itératif un algorithme récursif. Une application courante des piles est pour les calculs arithmétiques: l'ordre dans la pile permet d'éviter l'usage des parenthèses. Il existe trois possibilités pour représenter une équation (du moins de manière unidimensionnelle, les arbres en sont une généralisation bidimensionnelle), suivant la position relative des opérateurs et de leurs opérandes. Il faut avant tout définir l'arité d'une opération : une opération unaire nécessite un opérateur (− unaire, cosinus, log, factorielle...), une opération deuxaire (dénomination P. Trau, pour différencier d'une opération binaire qui pour moi traite des nombres en base 2) nécessite deux arguments (+, −, *, /, produit vectoriel,...), une opération ternaire nécessite trois opérandes (produit mixte,...). La notation préfixée (dite polonaise) consiste à placer l'opérateur, suivi de ses arguments. La notation postfixée (polonaise inverse) consiste à placer les opérandes devant l'opérateur. La notation infixée (parenthèsée) consiste à entourer les opérateurs deuxaires par leurs opérandes, pour les autres arités on place l'opérateur en premier, suivi de ses opérandes (entre parenthèses, séparés par des virgules pour les opérateurs ternaires). Les parenthèses sont nécessaires uniquement en notation infixée, certaines règles permettent d'en réduire le nombre (priorité de la multiplication par rapport à l'addition, en cas d'opérations unaires représentées par un caractère spécial (−, !,...). Les notations préfixée et postfixée sont d'un emploi plus facile puisqu'on sait immédiatement combien d'opérandes il faut rechercher. Détaillons ici la saisie et l'évaluation d'une expression postfixée (polonais) : on modifie le type de composantes de la pile (et donc les fonctions de base) :

```
#define operande 1
#define operateur 0

typedef struct
    {int type;
     union {float op_r;char op_c;}val;
    }composante;
void saisie(void)
```

```
/* marche pour toute notation puisque ne vérifie rien */
 {
  composante x;
  char txt[100],*deb;
  char rep;
  init_pile();
  printf("entrez opérandes (nombres) et opérateurs (+,−,*,/,C (cos),S)\n");
  printf("séparés par des blancs ou virgules\n");
  fflush(stdin);
  gets(txt); /* on lit un texte qui sera traité par après */
  deb=txt; /* deb pointe le début du texte non encore traité */
  do
   {
    while(*deb==' '||*deb==',') deb++;
    if(*deb==0)break;
    if(isdigit(*deb)||*deb=='.'||(*deb=='−'&&isdigit(*(deb+1))))
     {
      x.type=operande;
      sscanf(deb,"%f",&(x.val.op_r));
      empiler(&x);
      while(isdigit(*deb)||*deb=='.') deb++; /*pointer la suite*/
     }
    else /*cas d'un opérateur */
     {
      x.val.op_c=toupper(*(deb++));
      x.type=operateur;
      empiler(&x);
     }
   }
  while(1);
 }

float eval_post(void)
 {
  float r1,r2;
  composante x;
  if(depiler(&x)!=0) {puts("expression non postfixée");return(0);}
  if(x.type==operande) return(x.val.op_r);
  /* else inutile car les autres cas se finissent par return */
  /* on traite d'abord les opérateurs unaires */
  if (x.val.op_c=='C') return(cos(eval_post()));
  if (x.val.op_c=='S') return(sin(eval_post()));
  /* les deuxaires maintenant */
  r2=eval_post();
  r1=eval_post();
  switch (x.val.op_c)
   {
    case '+':return(r1+r2);
    case '−':return(r1−r2);
    case '*':return(r1*r2);
    case '/':return(r1/r2);
   }
  puts("erreur (code opératoire inconnu par exemple)");
  return(0);
 }

void main(void)
 {
  saisie();
  printf("l'expression postfixée vaut %6.1f\n",eval_post());
 }
```

Les piles permettent très souvent de clarifier l'implantation d'un algorithme, bien qu'évidemment on puisse utiliser le principe tout en n'utilisant que des tableaux ou listes.



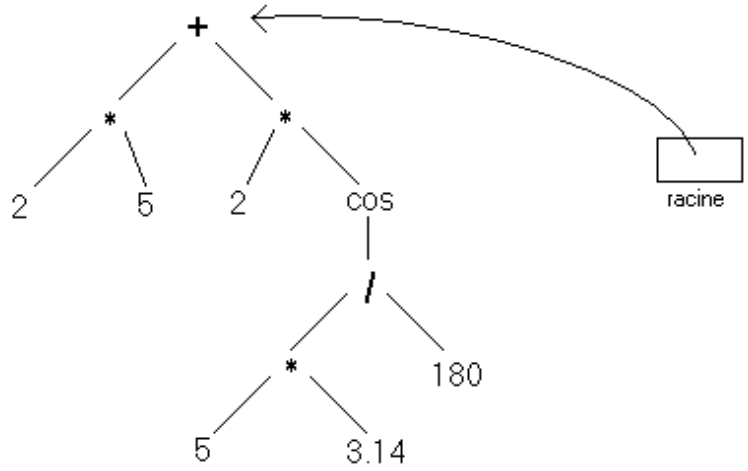
8 LES ARBRES

8.1 introduction

Les listes et piles sont des structures dynamiques unidimensionnelles. Les arbres sont leur généralisation multidimensionnelle (une liste est un arbre dont chaque élément a un et un seul fils). On utilise un vocabulaire inspiré des arbres généalogiques (sexiste il est vrai, bien qu'on utilise quelquefois les termes *fille* et *mère*). Chaque composante d'un arbre contient une valeur, et des liens vers ses *fil*s. On peut distinguer un *noeud*, qui est une composante ayant des fils, et une *feuille*, qui n'en possède pas (mais est fils d'un noeud) mais souvent on préfère utiliser un type noeud y compris pour les feuilles (tous les fils sont mis à NULL), surtout si l'arbre est évolutif (une feuille peut devenir un noeud, et inversement). Une *branche* rassemble un lien vers un fils mais aussi ce fils et toute sa descendance. Le noeud *racine* est le seul qui n'est le fils d'aucun noeud (tous les autres noeuds sont donc sa *descendance*). Comme pour le premier d'une liste, l'adresse de la racine est nécessaire et suffisante pour accéder à l'intégralité d'un arbre. Un arbre N−aire ne contient que des noeuds à N fils (et des feuilles). Une liste est donc un arbre unaire. De nombreux algorithmes ont été développés pour les arbres binaires (nous verrons que tout arbre peut être représenté par un arbre binaire). Comme les listes, les arbres sont complètement dynamiques, mais les liens sont également gourmands en mémoire. L'accès aux données est encore séquentiel, mais on verra qu'il reste néanmoins rapide.

8.2 expressions arithmétiques (arb_expr)

Nous allons détailler les fonctionnalités de base des arbres à l'aide d'un exemple : la représentation et l'évaluation d'expressions arithmétiques. Nous supposerons (voir chapitre sur les piles) utiliser uniquement des opérandes flottants, les opérateurs deuxaires +,−,*,/ et les opérateurs unaires C (pour cosinus) et S (sinus). L'expression 2*5 + 2*{cos[(5*3.14)/180]} se représentera :



Les composantes de l'arbre seront donc soit des opérateurs (noeuds), soit des opérandes (feuilles). Nous allons choisir les déclarations suivantes :

```
typedef float type_feuille;
typedef struct s_noeud_2
  {char op_c;
   struct s_comp*fils1;
```

```
   struct s_comp*fils2;
  }type_noeud_2;
typedef struct s_noeud_1
  {char op_c;
   struct s_comp*fils;
  }type_noeud_1;
typedef struct s_comp
    {int arite;
     union
         {type_noeud_2 n2;
          type_noeud_1 n1;
          type_feuille f;
         }val;
    }composante;
typedef composante *lien;
```

Nous prévoyons donc deux types de noeuds : à arité 1 (avec un fils) ou à arité 2 (deux fils). On aurait pu utiliser un seul type (à arité 2) en mettant NULL dans le second fils en cas d'arité 1. Une feuille par contre ne contient aucun lien. Une composante de l'arbre contiendra donc un indicateur de son type (ici l'arité) et, suivant ce type, soit une feuille, soit un noeud (d'arité 1 ou 2). Nous allons écrire une fonction créant un tel arbre (à partir d'une chaîne de caractères préalablement saisie, contenant l'expression en notation polonaise préfixée par exemple) :

`lien saisie_prefixe(char **deb)`

/* on donne une chaîne de char contenant l'expression en notation préfixée. les opérateurs possibles sont +,−,*,/(deuxaires), C,S(cos,sin,unaires). Les nombres sont séparés par des blancs (optionnels pour les opérateurs) Les nombres commencent par un chiffre (exemple 0.5 au lieu de .5).*/
/* deb pointe sur le début de chaîne, il pointera ensuite sur le reste de la chaîne (pas encore traitée) donc passage par adresse d'un pointeur */

```
{
 lien x;
 char c;
 while(**deb==' '||**deb==',') (*deb)++;
 if(**deb==0) /* on est arrivé en fin de chaîne */
     {puts("erreur : il doit manquer des opérandes");
      return(NULL);}
 x=(composante*)malloc(sizeof(composante));
 if(isdigit(**deb))
  {
   x->arite=0;
   sscanf(*deb,"%f",&(x->val.f));
   while(isdigit(**deb)||**deb=='.') (*deb)++;
  }
 else
  {
   c=toupper(*((*deb)++));
   if(c=='*'||c=='/'||c=='+'||c=='-')
    {
     x->arite=2;
     x->val.n2.op_c=c;
     x->val.n2.fils1=saisie_prefixe(deb);
     x->val.n2.fils2=saisie_prefixe(deb);
    }
   else if(c=='C'||c=='S')
    {
     x->arite=1;
     x->val.n1.op_c=c;
     x->val.n1.fils=saisie_prefixe(deb);
    }
   else printf("erreur, '%c'n'est pas un opérateur prévu\n",c);
  }
 return(x);
}
```

On peut remarquer que cette fonction est récursive. La création de l'arbre à partir d'une expression postfixée est similaire (scruter la chaîne dans l'autre sens par exemple), à partir d'une expression infixée c'est un peu plus dur, surtout si l'on ne veut pas imposer d'entrer toutes les parenthèses, par exemple 1+2+3 au lieu de (1+(2+3)).

L'utilisation d'un tel arbre devient maintenant très simple. On utilisera la récursivité. Pour évaluer l'expression, si c'est une feuille unique la valeur est celle de la feuille, si c'est un noeud, la valeur est obtenue en effectuant l'opération correspondante, après évaluation (récurrente, donc soit directement la valeur si feuille soit nouveau calcul) de ses fils :

```
float eval(lien x)
 {
  float r1,r2;
  if(x->arite==0) return(x->val.f);
  else if (x->arite==1)
   {
    if (x->val.n1.op_c=='C')
         return(cos(eval(x->val.n1.fils)));
    else if (x->val.n1.op_c=='S')
         return(sin(eval(x->val.n1.fils)));
   }
  else
   {
    r1=eval(x->val.n2.fils1);
    r2=eval(x->val.n2.fils2);
    switch (x->val.n2.op_c)
     {
      case '+':return(r1+r2);
      case '-':return(r1-r2);
      case '*':return(r1*r2);
      case '/':return(r1/r2);
     }
   }
  /* si aucun return n'a été effectué jusqu'ici*/
  puts("erreur (code opératoire inconnu par exemple)");
  return(0);
 }
```

Le parcours des arbres (c'est à dire le passage par tous les noeuds et feuilles) se fait d'habitude de manière récursive. On doit évidement parcourir l'arbre pour afficher son contenu. On peut utiliser dans le cas des arbres binaires trois possibilités de parcours : afficher la valeur du noeud puis récursivement de ses deux fils, afficher le premier fils, la valeur du noeud, puis le second fils, ou afficher les deux fils avant la valeur du noeud. On obtient dans notre cas directement les notations préfixé, infixée et postfixée (avec un petit travail supplémentaire pour la gestion des parenthèses en notation infixée, qu'on aurait pu simplifier si l'on avait accepté des parenthèses surabondantes) :

```
void affiche_prefixe(lien x)
 {
  switch(x->arite)
   {
    case 0:printf("%6.1f ",x->val.f);break;
    case 1:printf(" %C ",x->val.n1.op_c);
           affiche_prefixe(x->val.n1.fils);
           break;
    case 2:printf(" %c ",x->val.n2.op_c);
           affiche_prefixe(x->val.n2.fils1);
           affiche_prefixe(x->val.n2.fils2);
   }
 }

void affiche_postfixe(lien x)
 {
```

```
  switch(x->arite)
   {
    case 0:printf("%6.1f ",x->val.f);break;
    case 1:affiche_postfixe(x->val.n1.fils);
           printf(" %C ",x->val.n1.op_c);
           break;
    case 2:affiche_postfixe(x->val.n2.fils1);
           affiche_postfixe(x->val.n2.fils2);
           printf(" %c ",x->val.n2.op_c);
   }
 }

void affiche_infixe(lien x)
 {
  switch(x->arite)
   {
    case 0:printf("%6.1f ",x->val.f);break;
    case 1:printf(" %C( ",x->val.n1.op_c);
           affiche_infixe(x->val.n1.fils);
           putch(')');
           break;
    case 2:if(x->val.n2.fils1->arite!=0)putch('(');
           affiche_infixe(x->val.n2.fils1);
           if(x->val.n2.fils1->arite!=0)putch(')');
           printf(" %c ",x->val.n2.op_c);
           if(x->val.n2.fils2->arite!=0)putch('(');
           affiche_infixe(x->val.n2.fils2);
           if(x->val.n2.fils2->arite!=0)putch(')');
   }
 }
```

Notre exemple est un peu plus complexe que dans véritable arbre binaire, puisque nous acceptions également des noeuds à un seul fils.

8.3 listes triées

Il est possible de mémoriser une liste ordonnée à l'aide d'un arbre binaire. Chaque noeud de l'arbre contient une valeur et deux liens : l'un vers un sous−arbre ne contenant que des valeurs inférieures, l'autre vers des valeurs supérieures. Nous allons traiter cette fois des chaînes de caractères plutôt que des réels. Nous définirons donc les types suivants :

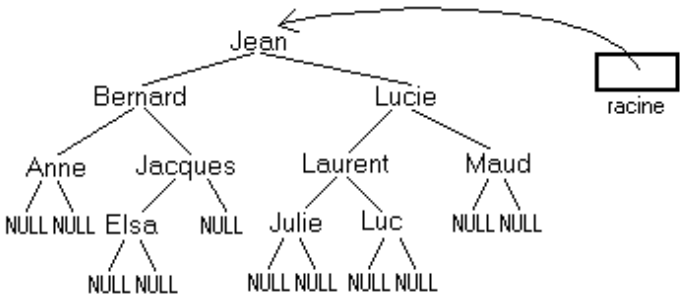
```
typedef struct s_noeud *lien;
typedef struct s_noeud
  {char *nom;
   lien fils1;
   lien fils2;
  }type_noeud;
```

Il vaut mieux ne pas prévoir de type spécifique pour les feuilles, l'arbre étant géré de manière évolutive (une feuille sera donc un noeud avec ses deux fils valant le pointeur NULL). Le nom contenu dans chaque noeud aurait pu être un tableau statique de caractères, mais pour optimiser le coût mémoire, nous utiliserons un tableau dynamique. Donc, pour allouer la mémoire d'un nouveau noeud, deux malloc seront nécessaires : un pour le noeud lui−même, et un pour le nom qui lui est associé. Ecrivons une fonction de création d'un noeud (on lui transmet le nom associé, elle retourne l'adresse allouée):

```
lien cree_feuille(char *nouv_nom)
 {
  lien x;
  int taille;
  x=(lien)malloc(sizeof(type_noeud));
  taille=strlen(nouv_nom)+1; /* compter le \0 */
```

```
  x->nom=(char*)malloc(taille);
  strcpy(x->nom,nouv_nom);
  x->fils1=NULL;
  x->fils2=NULL;
  return(x);
 }
```

Supposons entrer dans l'ordre : Jean, Bernard, Lucie, Anne, Jacques, Laurent, Maud, Luc, Julie, Elsa. Si vous créez progressivement l'arbre, vous pourrez voir que toute nouvelle valeur trouve toujours sa place, les feuilles se transformant petit à petit en noeuds, au fur et à mesure de l'augmentation de la taille de l'arbre.



La fonction effectuant la saisie et la création de cet arbre devra donc, après la saisie du nouveau nom à insérer, rechercher sa position définitive. Cette recherche sera récursive: sur un noeud donné, si le nouveau nom est plus petit, rechercher dans la descendance du fils1, sinon rechercher dans la descendance du fils2. La recherche s'arrête quand on arrive à un pointeur NULL (qui sera remplacé par l'adresse du nouveau noeud). En cas d'égalité, si l'on désire une création stable, un nouveau nom sera placé derrière ceux déjà existants. :

```
void insert_feuille(lien racine, lien x)
 {
  while(racine!=NULL) /* ou boucle infinie */
   {
    if(strcmp(x->nom,racine->nom)<0)
      if(racine->fils1==NULL){racine->fils1=x;return;}
      else racine=racine->fils1;
    else
       if(racine->fils2==NULL){racine->fils2=x;return;}
       else racine=racine->fils2;
   }
 }

lien saisie(void)
 {
  lien racine=NULL;
  char txt[100];
  do
   {
    printf("entrez un nouveau nom, @@@ pour finir : ");
    gets(txt);
    if(strcmp(txt,"@@@"))
      if(racine==NULL)racine=cree_feuille(txt);
      else insert_feuille(racine,cree_feuille(txt));
   }
  while(strcmp(txt,"@@@"));
  return(racine);
 }
```

On remarque l'efficacité de cette méthode : aucun déplacement d'élément, le tri se fait par insertion, mais avec recherche optimisée (du type dichotomie : l'espace de recherche est réduit à chaque test). Pour N insertions, on fait donc N recherches donc N*P lectures de composantes (P profondeur de l'arbre).

Une fois l'arbre créé, on peut afficher la liste triée par ordre alphabétique par un simple parcours infixé

(arbr_nom):

```
void affiche(lien x)
 {
  if(x!=NULL)
   {
    affiche(x->fils1);
    printf("%s, ",x->nom);
    affiche(x->fils2);
   }
 }
```

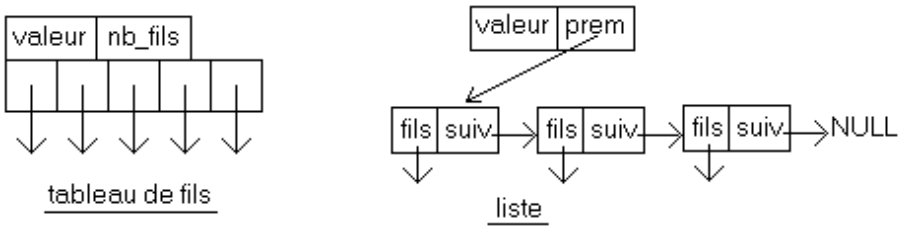
L'utilisation d'un arbre binaire pour une liste triée permet donc une programmation relativement aisée, des algorithmes rapides, mais moyennant un peu de place mémoire supplémentaire. L'arbre va posséder simultanément tous les avantages des autres structures de données:

- totalement dynamique: l'insertion d'un élément se fait sans aucun déplacement d'élément (ni même d'échange de liens comme c'était le cas pour les listes)
- rapide : la recherche d'un élément se fait en P tests maximum, P étant la profondeur de l'arbre (ce qui donne la même efficacité qu'une recherche dichotomique dans un tableau, dans le cas d'un arbre équilibré).
- parcours facile des données sans coût trop important en mémoire (avec deux liens on obtient une structure bidimensionnelle, alors qu'avec une liste à chaînage avant et arrière on garde une structure unidimensionnelle, bien que bidirectionnelle).

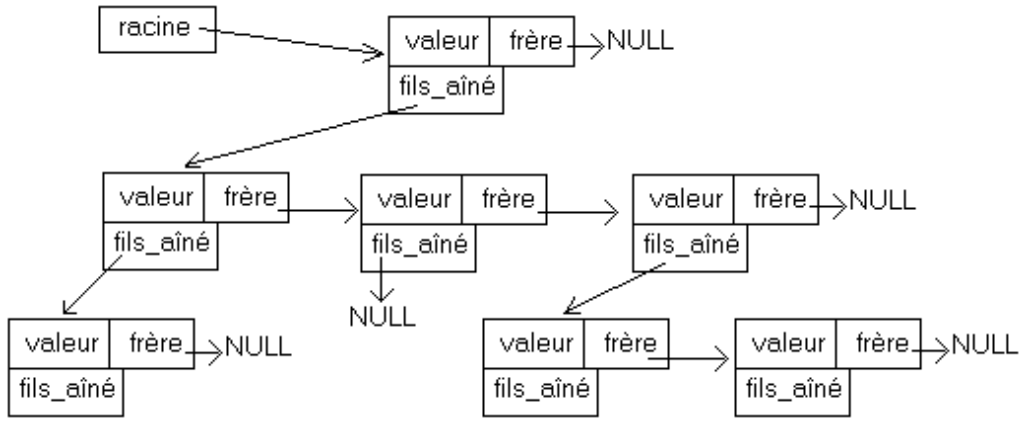
Par contre l'arbre binaire nécessite d'être équilibré pour profiter pleinement de ces avantages. Un arbre équilibré est un arbre organisé de telle manière à ce que sa profondeur soit minimale. A l'extrême, en cas d'introduction d'une liste de noms déjà triée, tous les fils1 pointeront sur NULL alors que les fils2 pointeront sur le suivant, on se retrouve dans le cas d'une liste chaînée simple. Différents algorithmes d'équilibrage d'arbres binaires existent, mais en général un algorithme simple permet un résultat satisfaisant (sauf en cas d'un très grand nombre de données), l'équilibre parfait n'étant pas nécessaire.

8.4 les arbres généraux

Quand le nombre de fils de chaque élément est variable, on peut soit prévoir un tableau statique des adresses des fils, soit prévoir un tableau dynamique, ce qui optimise l'occupation mémoire mais complique l'ajout de fils supplémentaires. Pour avoir une gestion parfaitement dynamique, on peut créer une liste des adresses des fils :

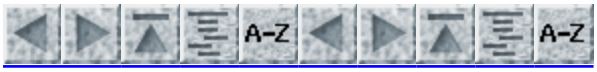


En fait, plutôt que de créer cette liste hors des noeuds, le plus simple (et qui utilise autant de mémoire) est d'associer à chaque noeud l'adresse de son fils aîné, et de son frère cadet. Accéder à tous les fils revient donc à accéder au fils aîné puis à tous ses frères:



On peut remarquer qu'un tel arbre est un arbre binaire: chaque noeud possède deux liens. On peut donc traiter tout arbre sous forme d'un arbre binaire.

Une autre amélioration possible d'un arbre est de permettre un accès toujours séquentiel mais bidirectionnel : pour un arbre binaire, chaque noeud possède en plus l'adresse de son père, pour un arbre généralisé ceci revient à ce que chaque frère connaît son aîné, l'aîné connaissant leur père commun. Cet accès bidirectionnel est coûteux en mémoire, mais complique également les modifications (plus de liens à gérer), pour par contre accélérer les parcours dans l'arbre. On ne prévoira cette amélioration que lorsque l'on utilise de fréquentes remontées (l'utilisation d'algorithmes récursifs ne nécessite en général pas ces liens, le fait de quitter une fonction appelée pour gérer un fils fait automatiquement retrouver le père).



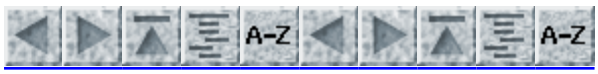
9 LES GRAPHES

Un graphe est en ensemble de noeuds reliés par des liens. Ce n'est plus un arbre dès qu'il existe deux parcours différents pour aller d'au moins un noeud à un autre. Un graphe est connexe lorsqu'il est possible de trouver au moins un parcours permettant de relier les noeuds deux à deux (un arbre est un graphe connexe, deux arbres forment un graphe non connexe). Un graphe est dit *pondéré* lorsqu'à chaque lien est associée une valeur (appelée *poids*). On utilisera un graphe pondéré par exemple pour gérer des itinéraires routiers (quelle est la route la plus courte pour aller d'un noeud à un autre), pour gérer des fluides (noeud reliés par des tuyauteries de diamètre différent) ou pour des simulations de trafic routier, pour simuler un circuit électrique, pour prévoir un ordonnancement dans le temps de tâches... Un graphe est dit *orienté* lorsque les liens sont unidirectionnels. Nous ne traiterons pas ici en détail les algorithmes associés aux graphes, mais nous aborderons quelques problèmes rencontrés.

On peut représenter un graphe de manière dynamique, comme les arbres (le nombre de liens par noeud est souvent variable). Une autre solution est de numéroter les N sommets, et d'utiliser une matrice carrée NxN, avec en ligne i et colonne j un 0 si les noeuds i et j ne sont pas reliés, ou le poids de la liaison sinon (1 pour les graphes non pondérés). Pour des graphes non orientés, la matrice est symétrique (par rapport à la diagonale), ce qui permet d'économiser de la mémoire mais peut compliquer les programmes (c'est la technique employée pour les éléments finis). Une représentation par matrice est surtout intéressante lorsqu'il y a beaucoup de liens (graphe presque complet), la représentation à l'aide de pointeurs étant moins gourmande en mémoire pour les graphes comportant peu de liens par noeud.

Un problème important est le parcours d'un graphe : il faut éviter les boucles infinies, c'est à dire retourner sur un noeud déjà visité et repartir dans la même direction. On utilise par exemple des indicateurs de passage (booléens), ou une méthode à jetons (on place des "jetons" dans différents noeuds bien choisis et on les fait suivre les liens). Une méthode souvent utilisée est de rechercher avant tout (et une seule fois) l'*arbre recouvrant* : c'est un arbre permettant de visiter tous les noeuds, n'utilisant que des liens existants dans le graphe (mais pas tous). Cet arbre recouvrant n'est évidement pas unique. En cas de graphe non connexe, il faut rechercher plusieurs arbres recouvrants.

On peut remarquer qu'un arbre recouvrant d'un graphe connexe à N sommets aura nécessairement N−1 liens. Pour les graphes pondérés, on peut rechercher l'arbre recouvrant de poids minimum (somme des poids des liens minimale). Différents algorithmes existent pour traiter ce problème.



10 LES FICHIERS

Les fichiers ont pour avantage d'être dynamiques (la taille peut varier ou du moins augmenter au cours de l'utilisation, du moins tant qu'il reste de la place sur le support), et de ne pas être volatiles comme l'est la mémoire (c'est à dire que les informations restent présentes en l'absence d'alimentation électrique). Leur principal inconvénient est le temps d'accès à une donnée (de l'ordre de 10^6 fois plus lent que la mémoire, en général plus de 10 ms pour un disque dur, mais jusqu'à plusieurs secondes pour une bande magnétique). L'insertion et la suppression d'éléments autre part qu'en fin d'un fichier est quasiment impossible, en général on devra passer par la création d'un nouveau fichier, y recopier le début du fichier initial jusqu'à la modification prévue, puis écrire les données modifiées puis recopier le reste du fichier. Les fichiers peuvent être séquentiels ou à accès direct.

10.1 les fichiers séquentiels

Un fichier séquentiel permet uniquement d'accéder aux données dans l'ordre de leur écriture. L'exemple le plus courant est la bande magnétique (ou streamer ou DAT). On n'utilise plus que rarement les fichiers stockés sur bande directement dans les programmes : du fait des possibilités des disques durs (jusqu'à plusieurs Giga−octets), on préfère utiliser des fichiers à accès direct, et les sauvegarder sur bande (séquentiellement) une fois le traitement terminé. Certaines grosses applications nécessitent néanmoins des traitements directement sur bandes magnétiques, il faut alors choisir les algorithmes en conséquence (en général dans ces cas on utilisera au moins deux lecteurs de bandes, un pour le fichier initial, un pour le fichier traité), en choisissant plus particulièrement des algorithmes divisant le fichier en sous−fichiers qui eux tiendront en mémoire centrale et pourront être traités plus rapidement (par exemple pour les tris, on utilisera par exemple un tri par fusion, surtout si l'on dispose de trois lecteurs : deux pour les sous fichiers à fusionner, le troisième pour le fichier final). Le grand avantage du fichier séquentiel est qu'il peut contenir des données de tout type, de taille différente (c'est désormais la seule raison de son utilisation). Pour stocker un graphique, on pourra enregistrer le nombre de segments, puis les caractéristiques de chacun des segments, puis le nombre d'arcs de cercles, puis leurs caractéristiques,...

On classe souvent sous la dénomination de fichiers séquentiels les fichiers de texte : se sont en fait souvent des fichiers à accès direct aux caractères (on peut accéder directement au Nième caractère), mais en fait l'accès est séquentiel au niveau des lignes : pour accéder à la Nième ligne, il faut lire le fichier depuis le début, jusqu'à compter N−1 signes de fin de ligne. Les fichiers de texte peuvent également être utilisés pour stocker des valeurs numériques : les nombres sont alors formatés pour être écrits en décimal à l'aide de caractères ASCII, et peuvent donc directement être imprimés ou réutilisés par un autre programme ou sur une autre machine, alors que la sauvegarde directe d'une valeur numérique est faite en binaire (recopie des 0 et 1 en mémoire), ce qui prend moins de place mais n'est plus lisible que par un programme qui utilise le même format.

Les fichiers séquentiels ne peuvent en général pas être modifiés, la seule possibilité étant l'ajout derrière les données déjà stockées (pour un fichier texte, sur disque, on peut remplacer des caractères mais pas en ajouter ni en supprimer sauf en fin de fichier).

10.2 les fichiers à accès direct

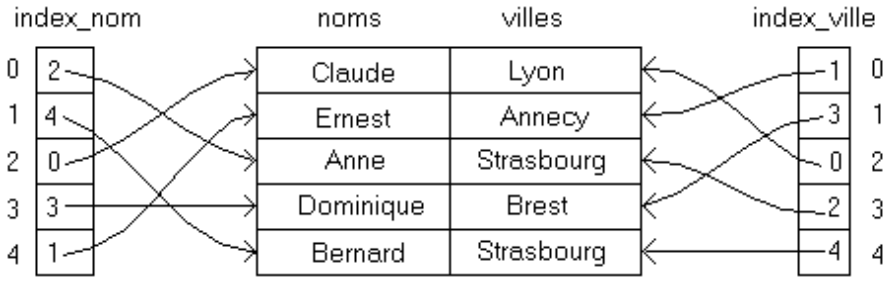
Un fichier à accès direct correspond à un tableau en mémoire : toutes ses composantes ont la même taille, on peut donc accéder directement à la Nième. Comme pour un tableau, les insertions et suppressions nécessitent des décalages des composantes suivantes, ce qui est en général trop long pour être envisageable. Ceci exclut donc l'utilisation de tout algorithme utilisant l'insertion (le tri par insertion par exemple), mais aussi ceux qui

déplacent des éléments sans les positionner à leur place définitive (le tri bulle par exemple). Une méthode de tri sera par exemple un tri par sélection et création d'un nouveau fichier. L'accès au données devra être optimisé : on préférera bien évidement la dichotomie à la recherche séquentielle, et même on préférera une dichotomie pondérée (avec estimation par calcul de la position probable), puisque tout calcul sera beaucoup plus rapide qu'une lecture dans le fichier.

Un fichier à accès direct peut également être utilisé pour stocker et traiter des listes, arbres ou graphes : on utilise la méthode du super−tableau : chaque élément est numéroté, en fonction de sa position dans le fichier, les liens étant ces numéros. Il faut bien garder à l'esprit qu'il ne faut jamais stocker sur fichier un pointeur : l'adresse où a été mise la valeur pointée a peu de chances d'être la même à la prochaine utilisation du programme. Ils est alors toujours nécessaire de numéroter les composantes et remplacer les pointeurs par le numéro de la valeur pointée avant de faire une sauvegarde sur fichier.

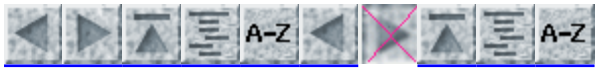
10.3 l'indexation

La méthode souvent la plus efficace pour l'utilisation de fichiers séquentiels est l'indexation (elle est également utilisable pour les autres types de données, stockées en mémoire). Les composantes sont stockées dans le fichier dans l'ordre de leur création. On utilise alors un tableau d'index, donnant en première position le numéro de la première composante, puis de la seconde,... L'avantage de cette méthode est que l'ajout de composantes est optimal : on rajoute la valeur en fin de fichier, on met à jour le tableau d'index. Tout déplacement d'une composante sera donc remplacé par une modification du tableau d'index, sans déplacement réel de la valeur dans le fichier. En général, ce tableau peut tenir en mémoire, ce qui permet une modification rapide, en général on préfère le sauver également sur support magnétique avant de quitter le programme, ce qui évitera de le recréer (par exemple refaire un tri) à la prochaine utilisation. On peut également utiliser une liste d'index si les déplacements sont fréquents (mais alors l'accès devient séquentiel). Le second avantage de cette méthode est que l'on peut utiliser simultanément plusieurs index : par exemple pour une liste de personnes, on peut créer un index pour le classement alphabétique des noms, un autre sur les villes, on accédera donc plus rapidement à tous les champs indexés, alors que les champs non indexés devront se satisfaire d'une recherche séquentielle, et ce sans modification dans le fichier (un tri par nom puis par ville auraient été nécessaires sans indexation). Par contre toute modification nécessitera la mise à jour de tous les tableaux d'index. Exemple:



La suppression, par contre, pose problème. En général, toujours pour éviter les décalages dans les fichiers, on préfère marquer d'un signe distinctif les champs supprimés (par exemple un nom non alphabétique ou vide), puis remettre à jour les index qui ne pointeront plus sur ce champ. Le retassage, assez long, n'est effectuée que sur ordre de l'utilisateur ou lorsqu'il quitte le programme. On peut aussi (comme dans la méthode du super−tableau) créer une liste des champs vides, ce qui permettra d'y accéder, plus rapidement que par une recherche séquentielle, lors de la prochaine insertion.

Sur un fichier indexé, on peut à nouveau se permettre des algorithmes utilisant l'insertion, puisque celle−ci n'affecte que l'index (à accès rapide). Pour un tri par exemple, on pourra utiliser le tri par insertion, à condition d'optimiser la recherche de la position d'insertion (par dichotomie pondérée par exemple), puisque celle−ci nécessite des lectures de champs dans le fichier alors que l'insertion n'entraîne que des décalages dans un tableau, d'une durée généralement négligeable devant le temps pris par la recherche. On peut également utiliser une liste d'index plutôt qu'un tableau si nécessaire.


A-Z
A-Z

11 CORRECTION DES EXERCICES

11.1 BOUCLE

```
#include <stdio.h>;
void main(void)
 {
  float puiss,x,result;
  printf("entrez x ");
  scanf("%f",&x);
  result=2 −5*x +(puiss=x*x) −3*(puiss*=x) +2*puiss*x;
  printf("résultat : %f\n",result);
 }
```

11.2 TUSEXO_A

```
#include <stdio.h>
#include "base_tus.inc"
#define dim 10
void main(void)
 {
  int i,nb;
  composante v,t[dim],moy=0;
  init_tus(t,&nb,dim);
  do
   {
    printf("entrez la %dième note (fin si <0 ou >20) :",nb+1);
    scanf("%f",&v);
    if(v<0||v>20) break; /* on aurait pu le mettre dans la condition du while */
    moy+=v;
   }
  while (!ajoute_val_tus(t,&nb,dim,v));
  if(nb) moy/=nb;
  printf("moyenne des %d notes : %5.2f\n",nb,moy);
  for(i=0;i<nb;i++) printf("%2dième valeur : %5.2f (écart/moyenne %+5.2f)\n",i+1,t[i],t[i]−moy)
 }
```

11.3 TUSEXO_B

```
#include <stdio.h>
#include <conio.h>
#include "base_tus.inc"
#include "mani_tus.inc"
#define dim 10
void lecture(type_tus t,int *nb)
 {
  composante v;
  do
   {
    printf("entrez la %dième note (fin si <0 ou >20) :",(*nb)+1);
    scanf("%f",&v);
    if(v<0||v>20) break; /* on aurait pu le mettre dans la condition du while */
   }
```

```
  while (!ajoute_val_tus(t,nb,dim,v));
 }

void main(void)
 {
  int nb;
  composante t[dim];
  char rep;
  init_tus(t,&nb,dim);
  lecture(t,&nb);
  do
   {
    printf("0 fin, 1 affiche, 2 rot gauche, 3 rot droite, 4 suppr, 5 ins :");
    rep=getche()−'0';
    printf("\n");
    switch (rep)
     {
      case 0:puts("au revoir");break;
      case 1:affiche_tus(t,nb);break;
      case 2:rot_gauche_tus(t,nb);break;
      case 3:rot_droite_tus(t,nb);break;
      case 4:{int pos;
              printf("position de l'élément à supprimer ? ");
              scanf("%d",&pos);
              suppr_tus(t,&nb,pos);
              break;}
      case 5:{int pos;composante v;
              printf("position de l'élément à insérer ? ");
              scanf("%d",&pos);
              printf("valeur à insérer ? ");
              scanf("%f",&v);
              insert_tus(t,&nb,dim,pos,v);
              break;}
      default:puts("code erroné !");
     }
   }
  while (rep);
 }
```

11.4 GAUSS_MRD

```
#include <stdio.h>
#include <alloc.h>
#include <math.h> /* pour fabs */

#define composante float
typedef composante *mat_dyn;

#define adr_mrd(l,c,nbc) ((l)*(nbc)+(c))

mat_dyn alloc_mrd(int nbl,int nbc)
 {return((mat_dyn)malloc(nbl*nbc*sizeof(composante)));}

void affiche_mrd(mat_dyn tab,int nblig, int nbcol)
 {
  int l,c;
  printf("\n");
  for(l=0;l<nblig;l++)
   {
    printf("|");
    for(c=0;c<nbcol;c++)printf("%5.1f ",tab[adr_mrd(l,c,nbcol)]);
    printf(" |\n");
   }
 }
```

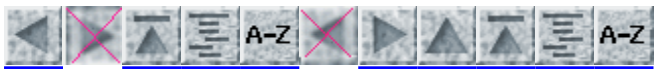
```
void autom_3_3(mat_dyn *t,int*nblig, int *nbcol)
 {
  composante ex[][3]={{1,1,−2},{1,3,−4},{−1,−2,6}};
  int l,c;
  *nblig=*nbcol=3;
  *t=alloc_mrd(*nblig,*nbcol);
  for(l=0;l<*nblig;l++) for(c=0;c<*nbcol;c++)
       (*t)[adr_mrd(l,c,*nbcol)]=ex[l][c];
 }
void autom_3_1(mat_dyn *t,int*nblig, int *nbcol)
 {
  composante ex[]={2,6,−1};
  int l,c;
  *nblig=3;*nbcol=1;
  *t=alloc_mrd(*nblig,*nbcol);
  for(l=0;l<*nblig;l++) for(c=0;c<*nbcol;c++)
        (*t)[adr_mrd(l,c,*nbcol)]=ex[l];
 }

void gauss_triangulation(mat_dyn A,mat_dyn B, int N)
 {
  int ce,l,c,lpivot;
  composante coef;
  for(ce=0;ce<N−1;ce++) /* ce=colonne à effacer */
   {
/* recherche du pivot le plus grand possible (dans les lignes qui restent)
*/
    lpivot=ce;
    for(l=ce+1;l<N;l++)
    if(fabs(A[adr_mrd(l,ce,N)])>fabs(A[adr_mrd(lpivot,ce,N)])) lpivot=l;
/* echange de la ligne du pivot et de la ligne ce (ne pas oublier B)*/
    for(c=ce;c<N;c++) /* tous les termes devant ce sont nuls */
     {
      coef=A[adr_mrd(ce,c,N)];
      A[adr_mrd(ce,c,N)]=A[adr_mrd(lpivot,c,N)];
      A[adr_mrd(lpivot,c,N)]=coef;
     }
    coef=B[ce];B[ce]=B[lpivot];B[lpivot]=coef;
    for(l=ce+1;l<N;l++) /* pour chaque ligne au dessous */
     {
      coef=A[adr_mrd(l,ce,N)]/A[adr_mrd(ce,ce,N)];
      A[adr_mrd(l,ce,N)]=0; /* normalement pas besoin de le calculer*/
      for(c=ce+1;c<N;c++)
        A[adr_mrd(l,c,N)]−=coef*A[adr_mrd(ce,c,N)];
      B[l]−=coef*B[ce];
     }
   }
 }

void gauss_resolution(mat_dyn A,mat_dyn B,mat_dyn X,int N)
/* on pourait rendre le résultat dans B (si l'utilisateur
désirait le garder, il lui suffit de le copier avant). Ceci
économise le tableau X et le tampon */
 {
  int l,c;
  composante tampon;
  for(l=N−1;l>=0;l−−)
   {
    tampon=B[l];
    for(c=l+1;c<N;c++) tampon−=A[adr_mrd(l,c,N)]*X[c];
    X[l]=tampon/A[adr_mrd(l,l,N)];
   }
 }

void main(void)
```

```
 {
  int nblA,nblB,nbcA,nbcB;
  mat_dyn A,B,X;
  autom_3_3(&A,&nblA,&nbcA);
  affiche_mrd(A,nblA,nbcA);
  autom_3_1(&B,&nblB,&nbcB);
  affiche_mrd(B,nblB,nbcB);

  if(nblA!=nbcA){puts("erreur fatale : matrice A non carrée");return;}
  if(nblB!=nblB){puts("erreur fatale : nb lignes du second membre incompatibles"); return;}
  if(nbcB!=1){puts("erreur fatale : le second membre doit avoir une seule colonne"); return;}

  gauss_triangulation(A,B,nblA); /*nblA=nbcA=nblB*/
  puts("résultat après triangulation : ");
  affiche_mrd(A,nblA,nbcA);
  affiche_mrd(B,nblB,nbcB);

  X=alloc_mrd(nblB,nbcB);
  gauss_resolution(A,B,X,nblA);
  puts("solution :");
  affiche_mrd(X,nblB,1);
 }
```

11.5 INSE_TTD

```
#include <stdio.h>
#include <alloc.h>
#include <string.h>
typedef char composante;
typedef composante *ligne;
/* ou composante ligne []*/
typedef ligne *mat_ttd;
/* ou ligne mat_ttd[] */

void affiche(mat_ttd tab,int nblig)
 {
  int l;
  for(l=0;l<nblig;l++)printf("%s\n",tab[l]);
 }

void lecture(mat_ttd *tdl,int *nblig)
 {
  /* on se limite à 100 lignes de 1000 caractères, mais une
fois sorti de la fonction, seule la place nécessaire est
réservée (variables locales). On supprime ces limites par
utilisation (toujours temporaire) de listes chaînées */
  composante txt[1000];
  ligne tab_de_lignes[100];
  int longueur;
  *nblig=0;
  printf("entrez vos lignes (fin du texte par '@@@' en déb de ligne):\n");
  do
   {
    gets(txt);
    longueur=strlen(txt); /* \0 non compris */
    if(!strcmp(txt,"@@@"))break;
    tab_de_lignes[*nblig]=(ligne)malloc(longueur+1);
    strcpy(tab_de_lignes[*nblig],txt);
    (*nblig)++;
   }
  while (1);
  *tdl=(mat_ttd)malloc((*nblig)*sizeof(ligne));
  memcpy(*tdl,tab_de_lignes,(*nblig)*sizeof(ligne));
 }
```

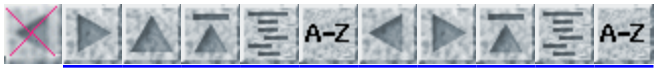
```
void tri_insertion(mat_ttd tab, int N)
 {
  int pt,dpg; /* position testée,dernier plus grand */
  ligne tampon;
  for(pt=1;pt<N;pt++)
   {
    dpg=pt−1;
    tampon=tab[pt];
    while(strcmp(tab[dpg],tampon)>0 && dpg>=0)
           {tab[dpg+1]=tab[dpg];dpg−−;}
    tab[dpg+1]=tampon;
   }
 }

void main(void)
 {
  int nbl;
  mat_ttd txt;
  lecture(&txt,&nbl);
  puts("avant tri :");
  affiche(txt,nbl);
  tri_insertion(txt,nbl);
  puts("après tri :");
  affiche(txt,nbl);
 }
```



12 Infographie

Vous trouverez ici un cours sur l'infographie (c'est à dire, ici, les algorithmes pour faire du graphique). On débute par les tracés les plus simples (le pixel, la ligne) pour les matériels de base, on passe aux cas plus complesxes, progressivement pour aborder le 3D, les surfaces complexes... Les algorithmes sont expliqués en clair, les exemples pratiques sont en C. Une bibliothèque graphique (en C) est accessible en téléchargement (gratuit).

Autres liens : Paul Bourke (en anglais)

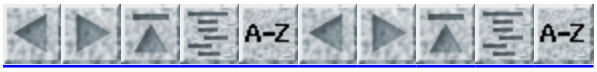


13 JUSTIFICATION

Dans de nombreux cas, des programmes généraux existent pour traiter nos besoins (en particulier en bureautique). Mais dans les autres cas, il est nécessaire d'écrire un programme spécifique au problème donné. Quand le problème est complexe et nécessite un programme bien réfléchi et optimisé, on accepte d'y passer du temps si la durée de vie du programme sera suffisante. Or les matériels évoluent aujourd'hui plus vite que les logiciels. Un logiciel créé sur PC devra peut−être plus tard tourner sur une autre machine (Mac ou station Unix par exemple). La seule solution pour prévenir ce problème est d'utiliser un langage standardisé (C, Pascal, Fortran,...) et de se limiter aux instructions standard, en évitant toute instruction spécifique au compilateur. Malheureusement les éditeurs de compilateurs ne précisent pas toujours clairement ce qui est normalisé et ce qui ne l'est pas. Par exemple chez Borland, tous les langages "Turbo" possèdent une fonction "clrscr" qui efface l'écran, tous les exemples de la documentation s'en servent, mais malheureusement aucun autre compilateur ne la connaît. La seule solution est de regrouper les fonctions non standard dans un fichier séparé bien commenté, pour n'avoir à modifier que ce fichier lors d'un changement de matériel.

Actuellement, on demande aux programmes d'utiliser le graphique, les sorties alphanumériques ne nous suffisent plus. Or aucun langage n'a d'instructions graphiques normalisées. Seul X Windows permet de transférer des programmes graphiques sur des matériels différents (mais pas tous, actuellement, surtout sur les stations), et de plus il n'est pas encore figé (et pas très simple). La meilleure solution est de se créer une librairie graphique, que l'on adaptera aux matériels utilisés.

De plus les algorithmes graphiques sont de parfaits exemples pratiques des possibilités d'optimisation d'algorithmes, les méthodes employées pour l'optimisation s'appliquant évidement à bien d'autres cas.



14 BIBLIOTHEQUE GRAPHIQUE DE BASE

Une bibliothèque graphique de base est un outil permettant d'accéder aux fonctions (point, droite, cercle...) de manière aisée. Elle n'a aucune connaissance des objets que l'on dessine (impossible d'éliminer les faces cachées par exemple).

14.1 tracés simples

14.1.1 matériel existant

Il existe deux grands types de matériels de visualisation : en mode matriciel (point) ou en mode vectoriel (ligne).

Un écran en mode vectoriel est (ou du moins était) composé d'une face fluorescente sur laquelle pointe un rayon que l'on dirige dans le tube par l'intermédiaire des 2 bobinages (comme un oscilloscope). On ne trace que ce dont on a besoin. Les tracés sont très précis mais aucune mémoire ne mémorise le dessin effectué. Le dessin reste sur l'écran jusqu'à effacement complet (modification et animation impossibles). Une amélioration a été apportée à ces écrans vectoriel : on peut effacer un trait en demandant de le dessiner en noir. Ces écrans donnent une définition (précision de tracé) excellente. Une droite inclinée y est cependant en général représentée par un escalier car la commande du rayon se fait numériquement (mais précisément).

Sur un écran en mode point (à balayage), le rayon balaye automatiquement l'écran ligne par ligne (on appelle mode entrelacé si pour éviter le scintillement on affiche les lignes paires puis les impaires). On ne commande plus alors que l'allumage ou non du rayon. Des circuits intégrés sont disponibles pour gérer ces écrans, il suffit de leur remplir une mémoire de 0 et de 1.

En mode non entrelacé avec des mémoires 8 bits, on allume le point à la ligne L, colonne C en mettant à 1 le $X^{ième}$ bit de la $Y^{ième}$ mémoire par le calcul : Position = (L*nb_col)+C; Y = partie entière de Position/8 ; X = reste de cette même division (en commençant L et C à 0). En général on donne les numéros de lignes croissants vers le bas car c'est le sens habituel du balayage.

Sur un écran à plusieurs niveaux de gris, chaque point est codé par un nombre (plusieurs bits) représentant

l'intensité du rayon. Sur un écran couleur, chaque point peut être représenté par 3 nombres : les intensités des trois couleurs de base (RGB : Red Green Blue). Cette méthode permet d'afficher simultanément toutes les couleurs, mais nécessite beaucoup de mémoire : si l'on choisit 256 niveaux possibles pour les trois couleurs de base, il faut 3 octets par point, près d'1 Mo pour 640*480 points. C'est la solution choisie pour les bons écrans couleur, mais avec au moins 1280 *1024 points sur l'écran. Quand la mémoire est plus limitée, chaque point sera représenté par un numéro de couleur, chaque couleur étant définie dans une table. C'est le cas de l'écran VGA : On possède 640*480 pixels, chaque point est représenté par son numéro de couleur, entre 0 et 15 (donc sur 4 bits) : il faut donc 300 ko de mémoire. Une table de 16 triplets définit chaque couleur : l'intensité des 3 couleurs de base RGB, chacune entre 0 et 63 donc sur 6 bits. On ne peut donc avoir simultanément à l'écran que 16 couleurs, mais dans une palette de 256000 couleurs. Au temps où les définitions d'écran et le nombre de couleurs étaient suffisamment faibles, la mémoire écran était directement accessible par le processeur (en CGA, à l'adresse B800h). Désormais, on délocalise la gestion de l'écran graphique (sur une carte ou un processeur), ce qui fait qu'on ne peut plus qu'envoyer une commande au processeur graphique, sans accès direct à sa mémoire. Ce dialogue est assez coûteux en temps, et dans ce cas, donner l'ordre de tracer une ligne ira bien plus vite que de donner un à un les ordres d'allumer les points de la droite. Sur certaines stations graphiques, le processeur graphique peut être plus volumineux que l'ordinateur lui−même (mais alors il peut être capable de traiter directement les projections 3D −> 2D).

On appelle pixel (picture element, en américain mal prononcé) un point de l'écran. Sur certains matériels (PC seulement ?) la taille d'un pixel n'est pas "carrée", s'est à dire que le nombre de pixels en X, sur une longueur donnée, n'est pas le même que sur Y pour la même longueur.

Sur papier, on peut disposer d'imprimantes matricielles (ou à jet d'encre) : Il faut réserver en mémoire une matrice de bits, y mettre des 1 aux endroits à "allumer" puis quand tout le dessin est prêt envoyer cette matrice sur l'imprimante. Pour une Epson 9 aiguilles par exemple, on n'utilise que 8 aiguilles en graphique. On devra envoyer un fichier contenant les 8 premiers bits de la première colonne, de la deuxième... jusqu'à la fin de la ligne, puis on recommence pour les 8 lignes suivantes... Attention, si par hasard on obtient un code binaire sur 8 bits correspondant à une commande de gestion de la liaison avec l'imprimante, on risque quelques problèmes (résolus sous MSDOS par copy /B). Les imprimantes laser (composées d'un rayon qui "dessine" sur un tambour de photocopieuse) sont en fait matricielles. mais elles ne permettent pas l'accès direct à leur mémoire. Elles possèdent un langage de programmation, soit spécifique, soit normalisé (PostScript) mais plus onéreux. On les commande donc comme un matériel vectoriel. Les tables traçantes fonctionnent dans un mode hybride : on peut se déplacer d'un incrément soit en X, soit en Y, soit les deux en même temps, ce qui donne des tracés beaucoup plus nets pour une même définition (même distance entre deux points consécutifs). Elles possèdent elles aussi un langage de programmation de type vectoriel, mais il n'est pas standardisé (le plus connu est HPGL).

L'entrée de coordonnées 2D se fait par la souris, le clavier, le joystick, la table à digitaliser, le crayon optique, à la rigueur la caméra. Mais en 3D, à part la machine à mesurer 3 axes, on ne sait pas bien entrer une coordonnée 3D autrement que par le clavier.

14.1.2 fonctions élémentaires

Pour un écran en mode matriciel, les seules fonctions dépendant de la machine nécessaires seront :

− l'initialisation du mode graphique (car par défaut on se trouve généralement en mode texte) ou ouverture d'une nouvelle fenêtre
− la fermeture du mode graphique
− l'allumage d'un pixel (et le choix de couleur)
− la possibilité de tester l'état d'un pixel (allumé ou éteint, ou sa couleur)

On en trouve un exemple sous Turbo C (versions DOS) dans bib_base.inc, ou en version DJGPP. Pour d'autres compilateurs (sur PC), une solution simple est d'utiliser l'int13 (description de Johan Calvat). Pour Xwindows, regardez la petite démo de Paul Bourke

L'accès à un pixel se fait soit par accès à la mémoire écran (il suffit alors de calculer l'adresse en fonction du type de balayage, des couleurs...), soit par une commande spécifique au matériel (une interruption logicielle sur PC). Tous les autres tracés pourront être écrits en langage standard, en utilisant ces fonctions élémentaires. Mais si l'on dispose d'un processeur graphique, l'appel direct de ces tracés par l'ordre adéquat sera souvent bien plus rapide.

Dans le cas d'une imprimante matricielle, il faut en général gérer en mémoire une matrice de pixels, et envoyer le tout à l'imprimante une fois tous les tracés effectués en mémoire. L'initialisation correspondra donc à une allocation de mémoire, et mise à 0 de celle−ci, la fermeture consistera en l'envoi du dessin à l'imprimante (avec ou sans mise en forme).

Dans le cas d'une table traçante (vectorielle), on allume un point en se déplaçant plume levée vers ce point, puis en abaissant puis relevant la plume (dans ce cas, il ne faudra pas décomposer les tracés en une suite de points).

14.2 représentation d'une droite (en mode point ou sur table traçante)

Pour une autre présentation de cet algorithme, regardez l'article d'Olivier Pécheux dans le n° 7 de PrograZine.

On peut évidement utiliser la formule Y=aX+b en travaillant sur des réels et en cherchant le point le plus proche du point théorique. C'est une très mauvaise solution car les calculs sur des réels sont très lents. De plus il faut traiter spécialement le cas des droites verticales. la meilleure solution est la suivante :

Soit à aller du point Xd,Yd au point Xf,Yf. Notons dX=|Xf−Xd| et dY=|Yf−Yd|.

Supposons dX > dY. Pour avoir une droite continue on devra donc essayer tous les X entre Xd et Xf pour déterminer leur Y et les dessiner. (dans le cas contraire faire le même raisonnement en échangeant X et Y).

$$\text{équation de la droite : } \frac{Y-Yd}{X-Xd} = \frac{Yf-Yd}{Xf-Xd} = \frac{dY}{dX} = \text{pente} = \text{dérivée}$$

A chaque avance de 1 en X, Y doit avancer de dY/dX (inférieur à 1). En prenant un compteur, initialisé à 0, à chaque avance de 1 en X, on doit y ajouter dY/dX. Une fois arrivé à une valeur entière, on augmente Y d'un pas. Puisque l'on cherchera maintenant la prochaine valeur entière, on peut aussi soustraire 1 au compteur et attendre que l'on atteigne à chaque fois 1.

Or je veux utiliser que des entiers. Modifions l'algorithme en multipliant le tout par dX : Initialisons un compteur à 0, à chaque avance en X ajoutons y dY. Lorsque le compteur arrive à dX, on augmente d'un pas en Y, on soustrait dX au compteur et l'on recommence.

Rq : En initialisant le compteur à 0, on reste toujours au dessous de la droite théorique. En l'initialisant à 1 (ou plutôt dX), on reste au dessus, en l'initialisant à dX/2 on est à cheval sur la droite théorique.

```
dX=|Xf−Xd|   dY=|Yf−Yd|
avX= +1 si Xd<Xf, −1 sinon
avY= +1 si Yd<Yf, −1 sinon
X=Xd   Y=Yd
allumer(X,Y)
si dX > dY alors
        +−−
        |compt=dX/2
        |pour I = 1 à dX faire
        |       +−−
        |       |X = X + avX
        |       |compt=compt+dY
        |       |si compt > dX alors
```

```
        |       |       +--
        |       |       |Y = Y + avY
        |       |       |compt=compt−dX
        |       |       +--
        |       |allumer(X,Y)
        |       +--
        +--
sinon
        +--
        |       (idem en échangeant X et Y)
        +--                                     exemple : base.inc
```

Sur écran, certaines droites auront un aspect désagréable : ce sont les droites proches de l'horizontale, la verticale ou la diagonale, pour lesquelles on obtient un aspect "d'escalier". Sur table traçante, ce problème est beaucoup moins visible pour deux raisons. En premier lieu parce que l'on peut déplacer la plume suivant X ou Y, mais aussi suivant la diagonale : les "marches" sont alors moins angulaires. Mais de plus le pas de déplacement de la plume est généralement très inférieur au diamètre du trait obtenu, les imperfections ne se distingueront donc pas (ou beaucoup moins).

14.3 autres courbes

14.3.1 par tâtonnements

Supposons connaître une relation simple entre x et y (pour un cercle x*x + y*y = R^2). Supposons le 1er point de la courbe connu (Xcentre+R,Ycentre) et le sens de la courbe (vers le haut). On représente les directions dans lesquelles on peut aller autour du point actuel par des chiffres entre 0 et 7 :

```
1  0  7
2     6
3  4  5
```

Essayons le point dans la direction choisie 0 : X,Y+1.

Or $X^2+(Y+1)^2$=R2' est supérieur à R2, donc essayons la direction suivante 1 :

$(X-1)^2+(Y+1)^2$=R2" est inférieur à R2. on a donc encadré R2. On va donc au point le plus proche de R2, et on débutera le calcul en ce point, avec la direction que l'on vient d'utiliser. On répète le procédé jusqu'au point final.

Remarques:

− Il faut évidement pour un cercle non centré en (0,0) X−Xcentre pour X et Y−Ycentre pour Y.
− Pour l'IBM PC, 640^2+200^2=449600 ne rentre pas dans un 16 bits, par contre on peut traiter des cercles jusqu'à un rayon de 46000 sur un entier 32 bits.
− On peut également utiliser la même méthode pour tracer une droite, mais l'algorithme est moins rapide que celui présenté plus haut.
− Sur un PC où les lignes sont plus espacées que les colonnes, il faut tracer une ellipse pour obtenir un cercle.

14.3.2 approximation par la tangente

La dérivée d'un cercle est dY/dX = − X/Y. Pour un pas (supposé assez faible) dX, on aura un déplacement en Y : dY = (−X/Y)*dX. Ceci n'est bien sur applicable que pour des portions proches de l'horizontale, sinon on travaille par pas sur Y. Cette méthode est très rapide et donne d'assez bons résultats, à condition de l'utiliser sur 45deg. au maximum puis utiliser les symétries (sinon on risque ne pas retomber sur le point de départ et donc dessiner des spirales). Voir cercles.c.

Rq : dY/dX = cste correspond à la méthode donnée plus haut pour une droite

14.3.3 formulation paramétrique

Pour un cercle, x=R cos[theta] y=R sin[theta]. En prenant des pas adéquats sur [theta] (en fonction du rayon), on trace le cercle par un ensemble de segments. Cette méthode nécessite évidement un calcul sur des réels. Elle est acceptable si on ne calcule pas les cos / sin à chaque fois (on se crée une table de sin sur 90deg., les autres valeurs en découlent rapidement).

La formulation paramétrique pour une courbe n'est intéressante que dans le cas d'un abscisse curviligne, c'est à dire quand un pas constant du paramètre conduit à une longueur constante de la courbe. Ceci donnera une précision constante sur toute la courbe.

14.4 remplissages / hachurages

Sur un périphérique en mode trait, on obtient le remplissage en resserrant les hachures, alors qu'en mode point on peut aussi utiliser un masque : une matrice de pixels (par exemple 16x16) où l'on donne les points à allumer et ceux à laisser éteints. Le masquage permet également des niveaux de gris sur un écran monochrome (évidement moins beau qu'un changement d'intensité de chaque pixel).

On peut utiliser deux approches totalement différentes :

14.4.1 remplissage d'une frontière déjà tracée

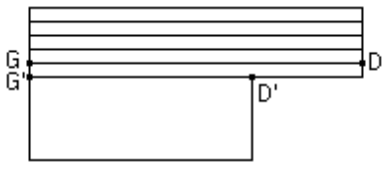
On donne un point de départ dans la frontière. On trace une droite vers la droite jusqu'à trouver un point allumé, idem vers la gauche. Puis on choisit un nouveau point de départ (le meilleur est en général le milieu du segment précédemment tracé), on descend d'un pixel, et s'il est éteint on recommence. Sinon on s'arrête et on fait de même pour la partie haute. Dans le cas d'un masquage, pour chaque point trouvé, on l'allume ou non suivant le masque et sa position. Dans le cas d'un hachurage, on se déplace de manière inclinée, on "descend" perpendiculairement d'une inter−hachures.

Ceci ne fonctionne que sur des frontières convexes et quelques autres cas particuliers. Mais ne désespérons pas, il existe une solution (et même plusieurs) :

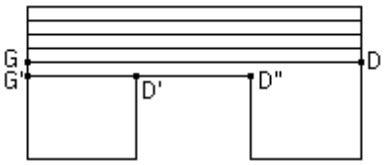
En traçant une ligne, on mémorise les deux points extrêmes G et D. En passant à la suivante, on obtient deux nouveaux points G' et D'. Si D et D' sont égaux (à un pixel près), il n'y a pas de problème. Si D'<D, alors il faut continuer jusqu'en D. Si tous les points (de D' à D) sont allumés, pas de problème. Sinon on mémorise le point où l'on était, on relance le même algorithme à partir de ce point (dans le même sens). Quand on aura traité entièrement le nouveau cas, on reviendra finir le point mémorisé. Par contre si D'>D, on continue sur la ligne précédente de D à D'. Si tous les points sont allumés, c'est bon. Sinon on mémorise et on traite à partir du premier point éteint (dans l'autre sens).

Mémoriser l'état actuel et répéter l'algorithme est facile à mettre en oeuvre grâce à la récursivité, mais on peut se passer de la récursivité en créant un tableau (géré sous forme de pile) des nouveaux points de départ à traiter.

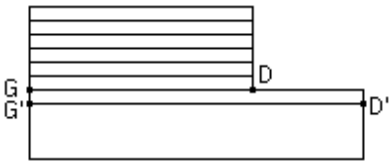
exemples : (j'allais en descendant)



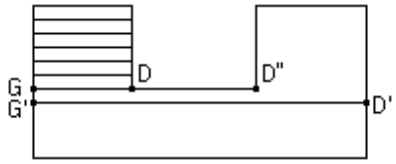
D'<D mais en allant de D' tous les points sont allumés, on continue donc normalement avec G=G' et D=D'



D'<D mais en allant de D' à D les points ne sont allumés que jusqu'en D"<D. On remplit (récursivement) à partir de D" (vers le bas suffit), puis on continuera normalement ligne suivante avec G=G' et D=D'



D'>D, mais en allant de D à D' sur la ligne précédente, tous les points sont allumés, on continue donc normalement avec G=G' et D=D'



D'>D, mais en allant de D à D' sur la ligne précédente le point D" est éteint. On remplit récursivement à partir de D" en remontant, puis on continue normalement avec G=G' et D=D'

Problèmes de cette méthode :

− impossible de recouvrir du dessin existant (algorithme du peintre)
− si le contour est mal fermé (un pixel éteint à une intersection, dû aux arrondis), on remplit tout l'écran.
− obligation de tracer les frontières.

14.4.2 frontière totalement définie

On donne au sous−programme le contour à remplir. Il pourra alors dessiner même sur d'autres dessins, on n'est pas obligé de tracer le contour. En général le contour est défini par un polygone (ensemble de points reliés par des droites), quitte à décomposer par exemple un arc de cercle en une succession de droites.

Si le polygone est CONVEXE, on commence au point le plus haut, en traitant toujours D et G (D=G au début sauf si plus d'un point à la même hauteur). Dans le polygone on pointe les deux directions à suivre. Puis on descend d'un pixel : Y. Du coté gauche du polygone, si $Y < Y_{point\_suivant}$ on calcule XG en fonction du segment actuel, sinon on passe au prochain segment pour calculer XG. On calcule de même XD, et on trace la droite de G à D. On progresse ainsi jusqu'à ce que l'on arrive au Y maximal (point le plus bas).

Pour les polygones NON CONVEXES, on peut trouver de nombreux algorithmes dans la littérature (leur nombre prouve leur complexité). En voici un :

On trace une droite fictive (infinie) parallèle à la direction de hachurage, et on détermine les points d'intersections avec tous les segments (finis) du polygone. On classe ces points dans l'ordre de leur position sur cette droite, puis on trace les segments reliant le premier au second point, le 3ème au 4ème,... On recommence à inter−hachures régulières. Les cas particuliers (points doubles, segments horizontaux,...) compliquent la méthode, on s'en sort en cherchant les intersections avec les segments du contour en incluant l'extrémité haute du segment et en excluant l'extrémité basse.

On peut également essayer de DECOMPOSER LE POLYGONE EN TRIANGLES (il existe plusieurs algorithmes, assez complexes), ou la METHODE A JETONS.

Rq : pour les hachures inclinées, suivant l'algorithme choisi il peut être plus intéressant soit de le généraliser soit de faire 2 rotations : sur le polygone pour traiter des hachures horizontales (angle [theta]) puis sur les hachures avant de les tracer (−[theta]).

14.5 les échelles

Il faut obligatoirement permettre à l'utilisateur de travailler à sa propre échelle. Une simple règle de 3 (2 car en X et en Y) le permet. De plus, en changeant uniquement le sous−programme de calcul d'échelle, on pourra exécuter le même programme sur des écrans différents (ou plus couramment sur un écran et une table traçante). En fait un bon programme utilisant du graphique n'utilise **jamais** une échelle dépendante du matériel (définition de l'écran) puisqu'il ne pourra jamais s'adapter à un changement d'écran, alors que le même programme doit souvent servir sur plusieurs ordinateurs n'ayant pas tous le même écran.

Rq : Tant qu'à calculer une échelle, autant proposer Y croissant vers le haut.

14.5.1 fenêtre objet et fenêtre papier

L'utilisateur travaille dans l'espace (infini), dans ses unités (n'importe quoi, des mm, daN, mètres de saucisson, semaines...), on les appelle **unités utilisateur** ou **unités objet**. Il ne veut en dessiner qu'une partie : il définit donc la **fenêtre objet** ou **fenêtre utilisateur**. En général il la définit par 4 valeurs : mini − maxi désirés en X et en Y (en unités utilisateur).

Mais on ne veut pas nécessairement mettre notre dessin sur la totalité de la feuille (ou écran). On définit donc une **fenêtre papier** ou **fenêtre écran** (on utilise le terme fenêtre papier même sur un écran). Ce sont les seules données qui ne seront pas fournies en unités utilisateur. La plupart du temps on les donne en unités écran (nombres de pixels), mais ceci rend le programme dépendant du périphérique utilisé, je préfère personnellement donner les limites de la fenêtre papier en pourcentage de la totalité de la feuille.

Si la fenêtre papier et la fenêtre utilisateur ne sont pas homothétiques, on obtiendra des distorsions (un cercle de l'espace utilisateur sera projeté dans la fenêtre papier sous forme d'une ellipse). Il est utile de proposer une échelle égale en X et Y (un carré reste un carré) et des échelles différentes (tracé de courbe par exemple).

14.5.2 clipping

Que faire si l'utilisateur demande de tracer une droite qui sort de la fenêtre ? Ceci arrive souvent : par exemple pour faire un zoom, plutôt que de multiplier toutes les coordonnées, on demande simplement un dessin en diminuant la fenêtre objet.

Pour le tracé d'un point, il faut effectuer 4 tests (comparer à Xmin, Xmax, Ymin, Ymax).

Pour les segments :

− Solution rapide mais peu satisfaisante : ne pas dessiner les droites dont une des extrémités sort de la fenêtre.
− Solution très lente : le calcul de la droite se fait complètement, on vérifie pour chaque point s'il se trouve dans la fenêtre avant de le tracer.
− Bonne solution :
On divise l'espace en zones (0000 est la fenêtre visible) :

1001	1000	1010
0001	0000	0010
0101	0100	0110

Chaque zone est définie par un bit : haut, bas, droite, gauche. Pour un point x,y on calcule : H=x>ymax, B=y<ymin, D=x>xmax, G=x<xmin.

On calcule le code HBDG pour les deux extrémités du segment. Si ces 2 codes sont 0000 on dessine le segment (les deux extrémités sont dans la fenêtre donc totalement visible). Sinon on calcule le code du segment qui est le ET entre les codes de ses extrémités. Si le code résultant est différent de 0, alors le segment est totalement invisible (soit totalement trop haut, trop bas, trop à droite ou à gauche) (c'est le cas le plus courant si les segments ne sont pas trop grands). Sinon on doit recalculer la (les) nouvelle valeur de l'extrémité dont le code n'était pas nul (intersection avec les bords de la fenêtre). Dans le cas d'un code de segment à 1 bit non nul, on sait directement sur quel bord est l'intersection, dans le cas à 2 bits non nuls, on doit tester une horizontale et une verticale.

Remarques :

− l'intersection d'une droite connue par 2 points et une verticale (x connu) ou une horizontale (y connu) est très simple (donc rapide).
− il est souvent pratique de posséder un sous programme de tracé de segment sans test de clipping (plus rapide) à utiliser quand on est sur de ne pas dépasser la fenêtre (et surtout l'écran, sinon on écrit n'importe où en mémoire). exemple : tracé du cadre.
− il est en général plus rapide de faire les tests en coordonnées écran (entiers), donc de tester le clipping sur la fenêtre écran et non la fenêtre utilisateur. Mais dans le cas de fenêtres non homothétiques, il faut décider si on limite sur la fenêtre papier ou sur la projection de la fenêtre objet.
− Cette méthode est unanimement reconnue comme la meilleure. En effet, elle traite de la manière la plus rapide possible tous les segment totalement visibles et presque tous ceux totalement invisibles. Seuls prennent plus de temps ceux qui risque d'être partiellement visibles, pour ceux−là on accepte ce temps supplémentaire puisqu'il nous permet de déterminer la partie visible du segment.

Pour les autres courbes, le test de clipping est souvent assez complexe et donne des résultats lents, on préfère donc en général l'approximer par un ensemble de segments, en définitive on obtient un résultat plus rapide

14.6 intersections

Les programmes graphiques ont souvent besoin de calculer de grands nombres d'intersections. Par exemple pour déterminer les faces cachées d'un dessin contenant 1000 segments (ce qui est peu, surtout si l'on approxime des courbes par de petits segments), il faudra calculer 1000^2 intersections de segments. De nombreuses études ont été menées sur ce sujet.

14.6.1 droite − droite

Comme précisé dans le tracé de segments, l'équation d'une droite est (Yf−Y)(Xf−Xd) = (Yf−Yd)(Xf−X), que l'on peut mettre sous la forme A.X + B.Y + C =0 (a, b, c du même type que les coordonnées, entier ou réel). Trouver l'intersection revient à résoudre deux équations à deux inconnues:

A .X + B .Y + C = 0
A'.X + B'.Y + C' = 0

DET = A.B' − A'.B, DET=0 si parallèles ou confondues (à epsilon près). Si droites colinéaires, on peut avoir pour des segments soit 0, 1 ou une infinité de points d'intersection. Sinon, l'intersection est X=(C'.B−C.B')/DET et Y=(C.A'−A.C')/DET (division réelle ou entiers). Il reste à voir si l'intersection se trouve DANS les deux SEGMENTS. De nombreux algorithmes existent, preuve que le problème n'est pas simple (du moins si l'on veut aller vite) :

− demi−plans :

équation d'un demi plan : A.X+B.Y+C < 0 (>0 pour l'autre demi−plan).

Il y a intersection si les deux extrémités de chaque segment sont de part et d'autre de l'autre droite, donc si :

(A.Xd' + B.Yd' + C).(A.Xf' + B.Yf' + C) <= 0 (de signe différent)

et (A'.Xd + B'.Yd + C').(A'.Xf + B'.Yf + C') <= 0

− **méthode vectorielle :** c'est le même principe :

les points A et B seront de part et d'autre de la droite CD : si $\overrightarrow{AC} \wedge \overrightarrow{AD}$ est de même signe que $\overrightarrow{BC} \wedge \overrightarrow{BD}$. Il faudra également vérifier que $(\overrightarrow{CA} \wedge \overrightarrow{CB}).(\overrightarrow{DA} \wedge \overrightarrow{DB}) \leqslant 0$.

Cette méthode permet de vérifier rapidement s'il y a intersection, mais ne donne pas le point d'intersection, à rechercher par la suite.

− **méthode paramétrique :**

X = Xd + (Xf−Xd).t_1 (1) X,Y appartient au SEGMENT df si 0<=t_1<=1

Y = Yd + (Yf−Yd).t_1 (2)

X = Xd'+ (Xf'−Xd').t_2 (3)

Y = Yd'+ (Yf'−Yd').t_2 (4)

(1)=(3) (Xf−Xd)t_1 + (Xd'−Xf')t_2 + Xd−Xd' = 0 (Cramer)

(2)=(4) (Yf−Yd)t_1 + (Yd'−Yf')t_2 + Yd−Yd' = 0

On résout pour trouver t_1 et t_2. On n'a intersection que si t_1 et t_2 entre 0 et 1.

Rq : dans le cas où l'on pense trouver beaucoup d'intersections, on choisit une méthode donnant rapidement le point d'intersection une fois que l'on a prouvé qu'il y a intersection (paramétrique par exemple). Par contre si l'on pense trouver peu d'intersection, on utilisera une méthode prouvant rapidement s'il y a intersection, quitte à refaire tout un calcul dans les rares cas d'intersection (vectorielle ou demi−plan puis résolution simple, sachant que le déterminant sera non nul).

− **intersections entre N droites :** on peut étudier toutes les intersections, il y a N(N−1)/2 possibilités (ordre N^2). Mais il existe d'autres algorithmes un peu plus puissants (ordre Nlog(N)), qui nécessitent de commencer d'abord par trier les points (en X croissant). Attention, le tri simple (chercher le plus petit, l'éliminer puis recommencer) prend N(N−1)/2 boucles et on n'a pas encore trouvé les intersections. Il faut utiliser des méthodes puissantes de tris. Ce type d'algorithme est surtout efficace dans certains cas particuliers (par exemple si les segments sont soit verticaux, soit horizontaux, par exemple pour gérer des fenêtres rectangulaires qui se chevauchent).

14.6.2 droite − cercle / cercle − cercle

supposons l'équation de droite Y=a.X+b (sauf verticale). L'intersection de cette droite et d'un cercle vérifiera donc $(X−xc)^2 + (a.X+b−yc)^2 = r^2$. Par substitution, on obtient :

$(1−a^2)X^2 −2.xc(b−yc).X + xc^2+(b−yc)^2−r^2 = 0$

On calcule le déterminant (0 racine double donc droite tangente au cercle, >0 si 2 points d'intersection, <0 si pas d'intersection) et l'on résout.

Dans le cas de deux cercles, on développe les 2 équations des cercles. En les soustrayant, on obtient l'équation de la droite passant par les 2 points d'intersection, on se ramène donc au cas précédent.

Rq : Ceci ne traite que les droites infinies et les cercles complets.

14.7 contraintes

On souhaite souvent définir des objets dépendant d'autres objets, en particulier en DAO :

− droite parallèle à une autre et passant par un point ou tangente à un cercle
− droite formant un angle avec une autre (90deg. par ex)
− droite tangente à un cercle et passant par un point
− droite tangente à 2 cercles (4 possibilités en général)
− arc de raccordement entre 2 segments
− cercle passant par 3 points ou inscrit dans un triangle
− cercle donné par le centre et passant par un (2 pour un arc) point
− cercle dont on donne le rayon ou le diamètre
− distances imposées (à une droite par ex)

Il est souvent nécessaire de proposer à l'utilisateur final toutes ces possibilités. Mais si le modèle (c'est à dire la représentation en mémoire des objets) ne permet qu'une seule définition par type d'objet (segment par 2 points par exemple), on traduit directement la contrainte (deux points de tangence). Mais une modification des "objets supports" (un cercle de tangence) ne modifiera pas l'objet (droite tangente). Il vaut mieux que le modèle mémorise ces contraintes, mais il en devient beaucoup plus complexe. Quelle que soit l'option choisie, les sous−programmes traitant les contraintes peuvent se trouver dans la bibliothèque de base, on s'en servira soit à l'introduction des données, soit uniquement en phase finale de dessin.

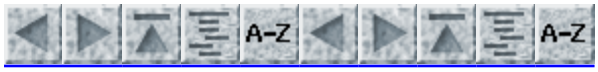
14.8 Conclusion

Une bonne bibliothèque de base contiendra différentes "couches". La couche 0 contient le minimum à connaître pour pouvoir faire des dessins (initialisation, allumer et tester un point, dans la couleur voulue). Changer d'ordinateur, de carte graphique ou d'imprimante nécessite la mise à jour de la couche 0 uniquement. La couche 1 contient les tracés de base (segment, cercles...) ne dépendant que de la couche 0. Si ces fonctions sont disponibles sur notre matériel (et plus rapides), on pourra évidement les remplacer, mais on gardera les versions standard en commentaire en cas de changement de matériel. On prévoira également, si possible, un tracé de segment horizontal et vertical plus rapides que le cas général, qui serviront dans les couches suivantes pour accélérer les algorithmes. Une couche 2 contiendra les remplissages dans le cas général : par test de l'écran et par définition du polygone frontière. Si le matériel dispose de ces fonctions (en 95 limité aux stations), on les utilisera. Mais sinon, on prévoira aussi les remplissages de convexes (du moins si l'implantation donne des résultats plus rapides dans ce cas), par exemple pour des dessins de surfaces décomposées en facettes toutes triangulaires ou quadrangulaires). On prévoira également dans la couche 0 un remplissage rapide de rectangle (côtés parallèles aux axes), utilisé entre autre pour l'effacement d'une fenêtre. Ce dernier cas seul est disponible sur cartes VGA, les autres cas devront donc être écrits en fonction des couches précédentes. Les fonctions des couches 0 à 2 ne devront pas être proposées aux utilisateurs (dépendent de la définition de l'écran), mais une couche 3 contiendra les fonctions de définition des fenêtres objet et papier, et tous ces tracés mais en coordonnées utilisateur. Cette couche contiendra également les tests de dépassement de fenêtres (clipping). On pourra ensuite rajouter des bibliothèques de fonctions utilitaires (contraintes, calculs vectoriels, transformations matricielles,...) qui ne sont pas directement liées aux couches précédentes.

Puisque les programmes sont aujourd'hui interactifs, on pourra également prévoir une bibliothèque d'entrée de données (clavier, souris, menus déroulants...)

Enfin, on prévoira des bibliothèques de fonctions spécifiques à un type d'utilisation, utilisant la bibliothèque

de base mais sans liens entre elles : bibliothèque 3D, tracés de courbes,....



15 TRANSFORMATIONS MATRICIELLES

15.1 représentation de fonctions planes par matrices 2x2

ATTENTION : les matrices sont représentées par des tableaux. En fait quand il y a deux bordures accollées il faut s'imaginer que l'on a à cet endroit la limite de la matrice (je n'ai pas le courage de tout passer en GIF).

soit	X'	=	a	b	*	X	donc	X' = aX + bY
	Y'		c	d		Y		Y' = cX + dY

On peut traiter certaines transformations dans le plan par cette méthode :

identité	1	0
	0	1

dilatation	Dx	0	si Dx et/ou Dy >1 on obtient une dilatation
	0	Dy	

symétrie	Sx	0	symétrie / (0,0) : Sx=Sy=−1. symétrie / axe X: Sx=1,
	0	Sy	Sy=−1. symétrie / axe Y : Sx=−1, Sy=1.

homothétie	1	Hxy	si Hxy ou Hyx =0 : cisaillement
	Hyx	1	homothétie si Hxy=Hyx

rotation	cos [theta]	−sin [theta]	rotation autour du point 0,0
	sin [theta]	cos [theta]	

remarques :

− le point (0,0) n'est jamais modifié.
− soit l'aire d'une surface S, après transformation :

```
     |a b|
S'=S*|c d|
```

On n'a donc pas de changement d'aire si le déterminant vaut 1 (symétrie, rotation...)
− matrice inverse : cas général :

a.X + b.Y = X' on cherche X,Y en fonction de X',Y'
c.X + d.Y = Y' par la méthode de Cramer
D=a.d−b.c (matrice non inversible si déterminant nul)
d'où X=(d.X'−b.Y')/D Y=(−c.X'+a.Y')

la matrice inverse de	a	b	est	1/D	*	d	−b	
	c	d				−c	a	

cas particuliers de matrices inverses :

dilatation : 1/Dx 1/Dy
symétrie : même matrice
rotation : −[theta]

application : symétrie par rapport à une droite faisant un angle [theta] avec Ox :

X'	=	cos [theta]	−sin [theta]	*	1	0	*	cos [theta]	sin [theta]	*	X		
Y'		sin [theta]	cos [theta]		0	−1		−sin [theta]	cos [theta]		Y		

On effectue d'abord une rotation des objets de −[theta], puis une symétrie / Ox, puis on retourne en position initiale (+[theta]). On obtient donc après multiplication des 3 matrices de transformation :

X'	=	$\cos^2$[theta]−$\sin^2$[theta]	2.cos[theta].sin[theta]	*	X	
Y'		2.sin[theta].cos[theta]	$\sin^2$[theta]−$\cos^2$[theta]		Y	

Remarque : le produit de matrices n'est pas commutatif mais associatif.

15.2 matrices 3x3 (coordonnées homogènes 2D)

La méthode précédente ne peut prendre en compte les translations. On utilise alors les coordonnées homogènes, c'est à dire que l'on représente un point par un triplet [ux,uy,u]. Ceci permet de dissocier la forme de la taille des objets. On se limite ici au cas u=1.

X'		a	b	tx		X		X' = aX + bY + tx
Y'	=	c	d	tx	*	Y	donc	Y' = cX + dY + ty
1		0	0	1		1		1 = 1

Une translation correspond à a=b=c=d=0, une transformation vue plus haut correspond aux mêmes a,b,c,d et tx = ty = 0. La troisième ligne de la matrice est obligatoire pour l'homogénéité des calculs, mais il n'est évidement pas nécessaire de la stocker ni de vérifier à chaque calcul la troisième équation (1 = 1).

matrice inverse : cas général, par la même méthode que ci−dessus, on trouve :

	a	b	tx			d	−b	ty.b−tx.d
inverse de	c	d	ty	= 1/(a.d−b.c)	*	−c	a	tx.c−ty.a
	0	0	1			0	0	ad−bc

Vous pouvez faire le produit pour vérifier. Le dernier terme de la diagonale vaut 1, la matrice inverse est également une matrice de transformation homogène.

Pour les opérations simples, on trouvera assez facilement l'inverse par l'interprétation physique.

15.3 transformations 3D

On utilise la même méthode mais avec des matrices 4x4 :

	X'		1	0	0	tx		X		
translation :	Y'	=	0	1	0	ty	*	Y		
	Z'		0	0	1	tz		Z		
	1		0	0	0	1		1		

	X'		dx	0	0	0		X		
dilatation :	Y'	=	0	dy	0	0	*	Y		
	Z'		0	0	dz	0		Z		
	1		0	0	0	1		1		

	X'		Ex	0	0	tx		X		
symétrie :	Y'	=	0	Ey	0	ty	*	Y		/(0,0,0) : Ex = Ey = Ez =−1
	Z'		0	0	Ez	tz		Z		/plan ij : Ei = Ej = 1, Ek =−1
	1		0	0	0	1		1		

	X'		1	0	0	0		X	
rotation /X :	Y'	=	0	c	−s	0	*	Y	c = cos [theta], s = sin [theta]
	Z'		0	s	c	0		Z	
	1		0	0	0	1		1	

rotation /Y :	X'	=	c	0	s	0	*	X	rotation /Z :	X'	=	c	−s	0	0	*	X
	Y'		0	1	0	0		Y		Y'		s	c	0	0		Y
	Z'		−s	0	c	0		Z		Z'		0	0	1	0		Z
	1		0	0	0	1		1		1		0	0	0	1		1

Matrice inverse : cas général :

X'		a	b	c	tx		X			X' − tx		a	b	c		X	
Y'	=	d	e	f	ty	*	Y		donc :	Y'− ty	=	d	e	f	*	Y	
Z'		g	h	i	tz		Z			Z' − tz		g	h	i		Z	

		1			0	0	0	1			1												

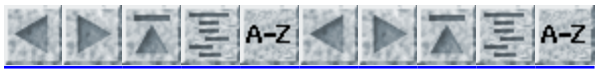
Le déterminant D de cette matrice 3x3 vaut donc D = $\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix}$ = a(ei−fh)−d(bi−ch)+g(bf−ec)

Si D=0, il n'y a pas de solution. La matrice n'est pas inversible car la transformation n'est pas bijective (le volume V initial est transformé en V'=D*V, vaut 0 en cas d'une matrice de projection 3D −> dimension inférieure).

si D différent de 0 alors

	X' − tx	b	c		a	X' − tx	c		a	b	X' − tx
X=1/D	Y'− ty	e	f	Y=1/D	d	Y'− ty	f	Z=1/D	d	e	Y'− ty
	Z' − tz	h	i		g	Z' − tz	i		g	h	Z' − tz

en développant les calculs et réorganisation sous forme X =a'X'+b'Y'+c'Z'+j' on obtient la matrice inverse.



16 PROJECTIONS 3D

Pour représenter un objet 3D tel que nous le voyons, il faut rechercher l'intersection des rayons partant de l'oeil, passant par les points de l'objet et arrivant sur une sphère (une droite se projette en un arc de cercle, d'ellipse ou de parabole, comme sur une photographie prise au "grand angle"). Dans la pratique on utilise un plan de projection, plus facile à gérer et aux résultats proches (une droite se projette en une droite).

16.1 parallèle

La méthode la plus simple est la projection parallèle. Elle est plus facile à opérer pour le dessin à la main mais ne rend pas un résultat conforme à la réalité, et n'est donc que peu utilisée avec les ordinateurs pour des rendus réalistes (une perspective ne nécessite pas beaucoup plus de calculs). Les rayons de projection sont tous parallèles et correspondent à une "vue de loin". Mais l'avantage de ces projections est que les projections de parallèles restent parallèles, et que les distances dans une direction donnée restent proportionnelles entre elles. Ce sera donc la projection utilisée pour les dessins techniques (mécanique en particulier).

Si le plan de projection est perpendiculaire à la direction de projection, on l'appelle projection axonométrique. Il suffit de tourner l'objet pour faire coïncider la direction de projection avec l'axe Oz, et ne plus dessiner que les x,y (z servant au problème des lignes cachées). Si les 3 axes sont raccourcis de façon égale (chacun à 120deg.), on l'appelle projection isométrique : rotation autour de Z de 45deg. puis de [theta] autour de Y (ou X) puis élimination de Z. [theta] ne vaut pas 45deg. car il faut ramener le point (1,racine(2)/2) sur l'axe et non (1,1)

	X'		1	0	0		cos T	0	sin T		R	−R	0		X	
	Y'	=	0	1	0	*	0	1	0	*	R	R	0	*	Y	avec R = racine(2)/2
	0		0	0	0		−sin T	0	cos T		0	0	1		Z	et tg T = racine(2)

Rq : comme sur un miroir, on obtient un repère indirect, il convient donc d'inverser un axe

On l'appelle dimétrique si deux axes sont réduits de façon égale, trimétrique sinon. Le cas du dessin industriel est un ensemble de projections orthogonales dimétriques sur les plans X=0, Y=0, Z=0 (un des axes est réduit à 0).

Si le plan de projection n'est pas perpendiculaire à la direction de projection mais à un des axes, c'est une projection oblique. Les deux plus connues sont la projection cavalière (les longueurs suivant les 3 axes sont conservées) ou cabinet (les fuyantes sont raccourcies de moitié). On représente généralement les fuyantes à 30 ou 45deg., notons C le cosinus de cet angle, S son sinus :

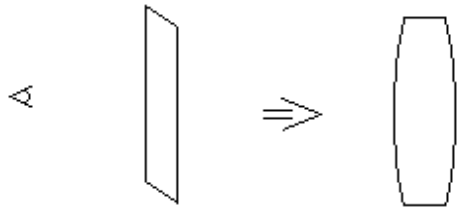
	X'		1	0	C			X		X'		1	0	C/2		X	
cavalière :	Y'	=	0	1	S	*	Y	cabinet :	Y'	=	0	1	S/2	*	Y	

	0		0	0	0			Z			0		0	0	0			Z	

16.2 perspective

16.2.1 méthodes

On pourrait utiliser les projections sphériques, correspondant plus à ce que l'on voit réellement. En effet un grand rectangle dans un plan perpendiculaire à la direction de vision devrait se projeter sous forme d'un tonneau puisque la partie centrale, étant plus proche de l'oeil, doit paraître plus large que les extrémités.

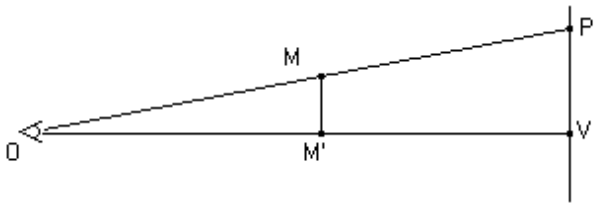


Sauf lorsque l'on recherche des effets spéciaux, comme par exemple l'effet obtenu en photographie par un objectif grand angle, on se limitera aux projections planes. En effet nous trouvons plus réalistes les projections qui gardent rectilignes les droites (mais par contre ne garde pas parallèles des droites qui l'étaient initialement, comme le font les projections parallèles). Il y a deux méthodes possibles pour implanter ces perspectives (qui reviennent au même) :

16.2.1.1 méthode vectorielle :

soient : O l'oeil, V le point visé (sur le plan de projection), M le point à projeter :

On calcule en premier lieu la longueur de OM', avec M' projection de OM sur OV, à l'aide d'un produit scalaire. Le rapport des longueurs OM' et OV donne le point projeté sur le plan P. Il reste faut maintenant, à partir des coordonnées dans ce plan dans l'espace, trouver les coordonnées X,Y sur l'écran.



16.2.1.2 méthode matricielle :

Il faut avant tout (à l'aide des matrices homogènes par exemple) déplacer la pièce (ou changer de repère), pour placer l'oeil en (0,0,0); puis la tourner pour placer le point visé sur l'axe Z (à la distance D), donc mettre V en position (0,0,D). Supposons M de coordonnées (x,0,z), et P en (x',0,D). Alors (d'après Thalès) : x'/x=D/z, donc x'=(D*x)/z.

Dans le cas général, on calcule la projection d'un point M (x,y,z) en un point P (x',y',D) en décomposant par les projections dans les plans Oxz et Oyz; donc x'=x.(D/z) et y'=y.(D/z)

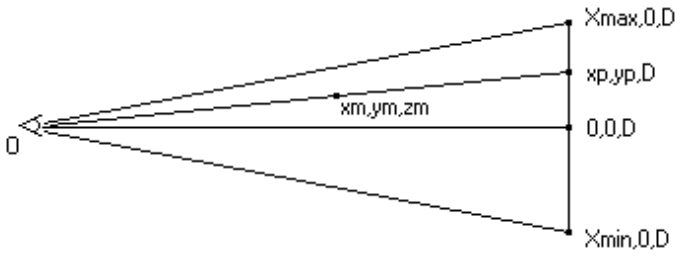
Pour mettre V, de coordonnées initiales (X,Y,Z), sur l'axe Z : effectuons une rotation de T autour de Z puis de P autour de Y. or $X^2+Y^2+Z^2 = D^2$ (coordonnées sphériques).

donc	cosP = D/R	sinP = racine(X^2+Y^2) /D
	cosT = X/racine(X^2+Y^2)	sinT = −Y/racine(X^2+Y^2)

Remarques pour les perspectives :
− modifier la distance oeil−pièce déforme la projection (ne pas se placer trop près), alors que modifier la distance oeil−plan de projection ne modifie que la taille de la projection, pas sa forme.
− la méthode vectorielle évite de déplacer la totalité de la pièce, mais le point projeté trouvé reste avec trois coordonnées, puisque dans un plan de l'espace, il faut encore faire la transformation 3D vers 2D, qui correspond soit à une transformation homogène, soit à deux produits scalaires, VP avec les deux vecteurs unitaires orthonormés du plan i et j.

16.2.2 clipping

On doit traiter une pyramide de vision, en général dans le repère déjà tourné pour effectuer la projection :



Le point M est visible si sa projection P est entre xmax et xmin, donc, d'après Thalès, xm/zm=xp/D, donc M est visible si :

xmin.zm/D <= xm <= xmax.zm/D et ymin.zm/D <= ym <= ymax.zm/D

Comme dans le plan, on peut tester la visibilité d'un segment en codifiant la visibilité des deux extrémités par 4 bits (haut, bas, droite, gauche).

Par exemple, calculons l'intersection d'un segment avec le plan supérieur :

$$\text{droite : } \begin{cases} x = t(x2 - x1) + x1 \\ y = t(y2 - y1) + y1 \\ z = t(z2 - z1) + z1 \end{cases} \quad \text{plan : } y = ymax.z/D = A.z$$

t(y2−y1)+y1 = A.t(z2−z1) +A.z1 donc t=(A.z1−y1)/(y2−y1−A.z2+A.z1)

ceci permet de trouver l'intersection (dans le segment si 0<=t<=1)

Rq : − Nos équations projettent aussi bien ce qui est devant l'oeil que ce qui est derrière. En DAO mécanique on n'a normalement rien derrière, ce qui n'est pas le cas en DAO architecture par exemple (on entre dans une pièce) ou simulation de vol (on évolue dans le décor, et de plus la visibilité peut être limitée). On doit alors traiter un tronc de pyramide de visibilité. On utilise un plan avant (par ex quelques cm devant l'oeil pour éviter les divisions par 0) et un plan arrière. On obtient deux conditions supplémentaires : zmin<=zm<=zmax. Il est facile de généraliser le clipping en utilisant 6 bits.

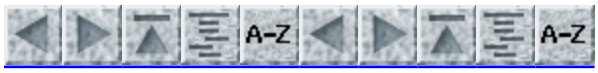
− On ne prévoit que de tester le clipping sur les segments. Ceci permet de ne pas tester tous les points générés, et de traiter facilement toutes courbes. Par exemple, en décomposant un cercle de l'espace en segments, uniquement projetés ensuite, on obtient automatiquement un tracé en ellipse sans écrire ni calculer nulle part son équation, ni besoins de tests de visibilité compliqués.

16.2.3 points particuliers

L'utilisateur donne le "parallélépipède utilisateur" (ses limites en x, y et z), les positions de l'oeil et le point visé (qui donne la direction de projection et la distance du plan). On calcule les projections des 8 coins extrêmes, ce qui permet de déterminer automatiquement une "fenêtre utilisateur" en 2D. L'utilisateur peut toujours donner une fenêtre papier, à condition que sa définition ne dépende pas des coordonnées utilisateur puisqu'il ne les connaît pas (en 2D).

Il faut évidement calculer les paramètres de projection (coefficients de la matrice) uniquement lors de la définition de la projection, et pas à chaque tracé de point ou de droite.

Remarque pour le tracé d'un cylindre en mode trait : on trace simultanément les deux cercles (par segments de droites en général), et on cherche les tangentes (points du cercle) les plus éloignées de l'axe. En traçant les cercles par pas de 10 degrés (36 tests), on obtient une précision suffisante (même si l'on trace plus précisément les cercles)



17 ELIMINATION LIGNES / SURFACES CACHEES

17.1 lignes cachées

Soit à représenter une courbe Z=f(X,Y), par exemple Z=sin(racine(X^2+Y^2)) qui définit une onde circulaire. Suivant le point de vue, on progresse en X ou Y pour commencer par les lignes les plus proches de l'oeil puis en s'éloignant. Comme pour un paysage montagneux, on utilise la méthode de la ligne de crêtes : la ligne de crêtes est la projection de la première ligne. Puis on étudie la ligne suivante. Segment par segment, soit le segment est projeté sous la ligne de crêtes, auquel cas il n'est pas dessiné, soit il est au dessus de la ligne de crête, on le dessine et on remet à jour la ligne de crête, soit il l'intersecte, on calcule alors l'intersection, on dessine à partir de l'intersection et on réaffecte la ligne de crête. On continue ainsi les tracés, la ligne de crête remontant progressivement.

− Dans le cas de courbes mathématiques, on utilise souvent une deuxième ligne de crêtes pour la limite inférieure.

− Plutôt que de calculer une ligne de crêtes exacte (formée d'un ensemble de segments difficiles à gérer du fait des problèmes d'insertions et de suppressions multiples), on peut se limiter à la définition de l'écran, c'est à dire créer un tableau donnant pour chaque pixel en Y la hauteur de la ligne de crêtes. Ceci n'est possible que si l'on possède le source de la bibliothèque de base (pour savoir quels pixels seront allumés).

17.2 faces cachées

Plusieurs algorithmes existent, mais aucun n'est parfait (en général très complexes pour prendre en compte les cas particuliers). Ces algorithmes fonctionnent avec des facettes planes (ou presque planes). Il est souvent plus facile de décomposer une surface complexe en facettes planes que de généraliser l'algorithme. Certains sont plus efficaces pour un dessin au trait (tracé des frontières de facettes), d'autres pour le tracé de faces colorées (remplissage des facettes). Les différents algorithmes ne peuvent être classés suivant leurs performance, car elles dépendent des cas traités (ou du moins des cas particuliers non traités).

17.2.1 surfaces orientées

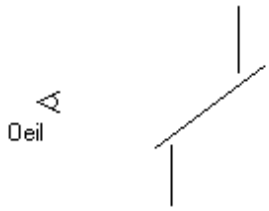
Si le volume à dessiner est convexe, en demandant à l'utilisateur de définir chaque face avec un "coté matière" (par un point central ou un "sens trigo"), on ne dessinera une face que si sa "normale matière" est dirigée vers l'arrière (son produit scalaire avec le vecteur Oeil−Visé positif).

Cette méthode est simple à mettre en oeuvre, mais est limitée aux volumes convexes, ce qui est assez rare. Par contre pour les volumes non convexe, bien que cette méthode n'élimine pas toutes les faces cachées, elle permet d'en éliminer rapidement la moitié.

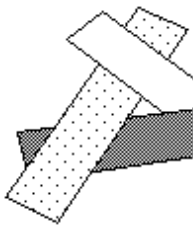
17.2.2 algorithme du peintre

En classant les facettes de la plus éloignée à la plus proche, il suffit de les tracer dans l'ordre. Un tracé au trait est impossible (à moins de remplir les faces avec la couleur de fond, ce qui est impossible avec une table traçante), et le dessin est long (on trace et colorie un même point de l'écran autant de fois que de faces superposées). Le remplissage de surfaces doit fonctionner pour pouvoir "passer" sur les tracés précédents et non s'arrêter contre.

Mais le véritable problème est le classement des facettes. Dans certains cas la position relative de deux surfaces dépend d'autres surfaces :



Pour que l'algorithme du peintre fonctionne, il faut tracer dans ce cas en dernier la face la plus éloignée. En effet, il ne faut comparer la position que de surfaces dont les projections se superposent.



Cet exemple montre qu'il existe des cas où l'algorithme du peintre ne peut s'appliquer. La seule solution dans ce cas est de couper une des faces en deux. Dans le cas où les faces sont décomposées en petites facettes planes, cet algorithme donne de très bons résultats (utilisé dans de grands codes d'éléments finis par exemple).

17.2.3 calculs de facettes

Pour chaque facette, on compare sa projection avec celles de toutes les autres facettes. Si les projections ont une intersection (prévoir aussi le cas d'un face cachant entièrement la facette), regarder la droite de projection passant par le point d'intersection, déterminer les points d'intersection avec les deux faces, modifier la frontière de la face cachée. Une fois les calculs terminés, on peut tracer la partie non cachée de la facette, s'il en reste une. Le tracé est rapide, mais le calcul long. Les calculs sont complexes (programme touffu), les cas particuliers nombreux (face cachée coupée en plusieurs sous−facettes visibles, une même arrête coupant plusieurs fois la surface...). Si l'on désire colorier les facettes, il faut un algorithme pour polygones non convexes et avec "trous". la meilleure preuve de la complexité de sa mise en oeuvre est l'existence des autres algorithmes

17.2.4 élimination des arrêtes cachées

Pour chaque arrête : on teste sa visibilité par rapport à chaque facette. Soit elle est complètement cachée, on abandonne alors cette arrête. Soit elle est complètement vue (parce qu'elle est "devant", c.a.d dans le même demi−plan que l'oeil, soit sa projection est à l'extérieur de la projection de la facette) on passe à la prochaine facette (ou on dessine l'arête si toutes les facettes ont été traitées). Soit elle est partiellement cachée, on ne garde que la partie de l'arrête vue (qui peut former plusieurs nouveaux segments) dont on testera la visibilité par rapport aux facettes restantes. Cet algorithme n'est possible que pour un dessin au trait, le remplissage en couleurs des différentes facettes n'est pas possible avec cet algorithme. Si le modèle le permet, on gagnera du temps en ne traitant qu'une fois un segment faisant partie de deux facettes.

17.2.5 tubes de projection

On divise la pyramide de vision en 4. Pour chaque nouvelle pyramide : soit aucune facette n'y entre, on abandonne alors la pyramide (rien à tracer). Soit une facette dépasse tout autour de la pyramide et elle est en avant des autres facettes qui entrent (tout ou en partie) dans la pyramide, on dessine toute la base de la pyramide dans la couleur de la facette (ou rien en cas de dessin au trait) et on l'abandonne. Dans tous les autres cas on partage la pyramide en 4, et on recommence. Lorsque l'on arrive à une pyramide de la taille d'un pixel, on allume le pixel de la couleur des frontières de facettes et on abandonne la pyramide (pas la peine d'être plus précis que le matériel). Cet algorithme est rapide quand il y a beaucoup de faces cachées et que le périphérique n'est pas trop précis. Le tracé est fait point par point (donc impossible sur table traçante) dans un ordre qui pourra paraître original à celui qui regarde la progression du dessin.

17.2.6 plans de balayage

On fait passer un plan dans la scène correspondant à une droite de balayage de l'écran et contenant l'oeil. On analyse l'ensemble des intersections des facettes avec ce plan, ce qui nous donne un ensemble de droites coplanaires (avec pour chacune la couleur de la facette dont elle est issue). Il faut alors déterminer les parties vues et les parties (lignes) cachées dans le plan, ce qui est plus facile que dans l'espace. On utilise autant de plans que le permet la définition de l'écran.

Cet algorithme se prête bien aux problèmes de transparence, et du fait de sa similitude avec le matériel, au câblage ou la programmation de l'algorithme directement dans la carte graphique.

17.2.7 rayons de projection

En partant de chaque pixel de l'écran on trace un rayon jusqu'à l'oeil, on calcule toutes les intersections entre les faces et ce rayon, puis on choisit la couleur de la surface la plus en avant. Ceci donne de longs calculs mais se prête au parallélisme et au câblage.

17.2.8 éclairage de surfaces

On peut donner une impression de réalisme à un dessin 3D en utilisant des niveaux d'éclairage. On suppose dans un premier temps une source lumineuse située dans la scène (si possible pas dans l'axe de projection). Chaque facette reçoit une couleur d'autant plus claire que la normale à cette surface est proche du rayon venant de la source lumineuse et traversant la surface (produit scalaire). Les résultats sont encore meilleurs avec plusieurs sources lumineuses, en prenant la somme des luminosités. Si l'on utilise des facettes qui peuvent être grandes, il faut soit calculer une luminosité variant sur la surface, soit découper en petites facettes (si les pas en couleurs sont assez faibles, le résultat sera satisfaisant). Pour un résultat vraiment réaliste, il faudrait en plus vérifier si les rayons lumineux ne sont pas coupés par une autre partie de la scène, et créer des ombres. Les calculs sont alors très longs, en général on triche en utilisant des sources lumineuses n'éclairant pas trop loin (l'intensité baisse avec la distance) et un choix manuel judicieux des différentes sources (assez nombreuses).

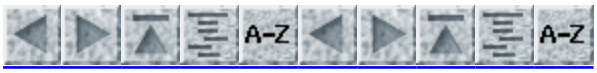
17.3 problème des surfaces gauches

Soit on prévoit divers types de surfaces particulières (on prévoit les cylindres et sphères par exemple), soit on décompose en facettes planes, soit on essaie de trouver une formulation mathématique permettant de modéliser les surfaces gauches (appelées à tort surfaces non mathématiques). On se limite généralement à des approximations de surfaces suffisamment précises (y compris pour l'usinage). Elles correspondent en général à l'extension 3D des problèmes de lissage de courbes (avec des principes mathématiques plus puissants d'une dimension).

Après le problème de la représentation mathématique de ces surfaces, le premier problème est celui de la

détermination des tangentes. Les cas particuliers se traitent surtout empiriquement (j'ai traité plus haut les cylindres), dans le cas général, après rotation de la surface correspondant au point de vue, on cherche l'ensemble des Z max (type ligne de crêtes) soit par tests itératifs, soit par dérivation (dérivée seconde nulle). On peut aussi utiliser un méthode de type "division de pyramide" pour traiter la surface.

Autre problème rencontré, les portions de surface cachées. Une solution couramment employée consiste à décomposer la surface en petites facettes planes. Pour une visualisation acceptable, les erreurs dues à la décomposition en facettes doivent être inférieures à la définition de l'organe de tracé.



18 COURBES ET SURFACES

18.1 introduction

On a souvent besoin de représenter dans l'ordinateur des courbes ou surfaces dont le type n'est soit non classifiable (ni parabole ni cercle...) soit non connu à l'avance. On les regroupe généralement sous le nom de "courbes et surfaces non mathématiques", car l'équation mathématique n'est pas connue par le programmeur. Ces courbes ou surfaces sont en général définies par un ensemble de points, plus ou moins nombreux.

Le problème qui se posait à la Régie Renault consistait à représenter les surfaces de carrosserie dessinées par les designers, pour réaliser les matrices d'emboutissage. La pièce était définie par des sections planes successives. A l'arrivée des machines à commande numérique, la méthode manuelle a dût être abandonnée. Après création d'une maquette 3D (en bois, plâtre,... modifiée au fur et à mesure), la surface était déterminée par une matrice de points obtenus à l'aide d'une machine à mesurer 3D. Pour obtenir une bonne précision, le nombre de points nécessaire est très important. Le nombre de maquettes réalisées en cours de conception étant important (d'où pertes de temps), on a demandé aux stylistes de créer sur ordinateur, les maquettes pouvant alors être facilement et rapidement réalisées en commande numérique. P. Bezier a du mettre au point une méthode permettant de définir la surface par un nombre minimal de points caractéristiques, permettre de modifier facilement la surface par déplacement d'un minimum de points, pouvoir représenter toute surface (y compris plane), sans "cassure" (continûment dérivable).

18.2 courbes

18.2.1 représentation des points

Une courbe peut être représentée par :

− son équation explicite (y=f(x) dans le plan, deux équations dans l'espace par exemple y=f(x) z=g(x), ce qui permet de limiter cette étude au cas plan),

− une définition paramétrique : x(u), y(u), z(u), u entre 0 et 1 ou abscisse curviligne,

− une définition paramétrique vectorielle :

$$\vec{P}(u) = f_0(u).\vec{a_0} + f_1(u).\vec{a_1} + \ldots + f_n(u).\vec{a_n}$$

18.2.2 polynômes de Lagrange

Soient N+1 points connus. On cherche à définir une courbe passant exactement par ces points Pi de coordonnées (Xi,Yi), i=0,1,2,...n.

On définit $Y=Pn(x) = \sum_{i=0}^{n} Li(x).Yi$

Pour que la courbe passe par les points imposés, on peut choisir les polynômes de Lagrange Li tels que :

Li (x) = 1 si x = Xi , Li (x) = 0 si x différent de Xi

Il suffit de prendre :

$$Li(x) = \frac{(x-X0)(x-X1)...(x-Xi-1)(x-Xi+1)...(x-Xn)}{(xi-X0)(xi-X1)...(xi-Xi-1)(xi-Xi+1)...(xi-Xn)}$$

Cette méthode est simple à mettre en oeuvre. Les coefficients sont d'autant plus longs à calculer que l'on a de points. On ne peut pas imposer de condition de tangence, ni influer sur la forme de la courbe entre les points imposées (on peut obtenir des oscillations importantes). Si les nombres de points sont importants, on peut obtenir des produits de Xi−Xj dépassant la capacité de la machine, et prenant beaucoup de temps de calcul. On peut étendre le principe en donnant les points de passage et leurs tangente (polynôme de Hermite). On peut aussi, pour limiter le nombre de points, interpoler la courbe par morceaux. Par Lagrange, la courbe obtenue n'est pas continûment dérivable (rupture de pente entre morceaux) par Hermite seule la dérivée première est continue.

18.2.3 splines

Au lieu de choisir parmi les polynômes de degré n+1, passant par nos n+1 points, un polynôme tel qu'il soit facile à déterminer (Lagrange), choisissons celui qui minimise la fonctionnelle l_q (s) :

$$L_q(s) = \int_{Xo}^{Xn} \left[\frac{d^q y}{dx^q} \right]^2 dx$$

Si q=2, cette fonctionnelle représente l'énergie de déformation en flexion d'une poutre. La minimiser revient à faire passer une fine règle de bois (spline en anglais) par des points imposés. Une autre interprétation consiste à dire que l'on veut minimiser les variations de la tangente à la courbe.

Remarque : la dérivée d'ordre (2q−2) est continue. On peut donc facilement approximer la courbe par morceaux.

la solution est de la forme :

$$s(x) = \sum_{q-1}^{n}$$

$$\sum_{j=0} a_j . x^j + \sum_{i=0} P_i(x)$$

$$\text{avec } Pi(x) = \frac{di <x-xi>_+^{2q-1}}{(2q-1)!}$$

et <x−xi>+ = 0 si x < xi , <x−xi>+ = (x−xi) si x >= xi (partie positive)

Les inconnues du problème sont a_j et d_i .

or pour i=0,1,...,n on a : $s(x_i) = y_i$

$$\text{et } \sum_{i=0}^{n} d_i x_i^k = 0 \text{ pour } k=0,1,...,q-1$$

18.2.4 courbes de Bezier

−>

−>
−>
−>
soit
P(u) =
OA + f_1 (u)
a_1 + ... + f_n (u)
a_n , avec u dans [0,1]
Cette définition de la courbe est possible aussi bien en 2D qu'en 3D. Les vecteurs a_0, a_1,..., a_n forment le "polygone caractéristique".

18.2.4.1 conditions

on impose à l'origine (u=0) :
− passage en A (f_0 =1 , f_i (0)=0 pour i > 0)
− la tangente ne dépend que de a_1 (f'i (0)=0 pour i>1)
− la dérivée seconde ne dépend que de a_1 et a_2 (f''i (0)=0 pour i>2)
− de façon générale, la dérivée kième ne dépend que des k+1 vecteurs a_o à a_k

De plus, le dernier sommet du polygone caractéristique impose le point final de la courbe (u=1), la tangente en ce point ne dépend que du dernier vecteur a_n, etc...

Exemple : pour n=3 : On impose les conditions suivantes :

− passage de la courbe aux deux extrémités du polygone caractéristique : $f_1(0)=f_2(0)=f_3(0)=0$; $f_1(1)=f_2(1)=f_3(1)=1$

− la tangente en A ne dépend que de a_1 (idem à l'autre extrémité) :

$$\frac{d}{du} f2(0) = \frac{d}{du} f3(0) = 0 \quad \frac{d}{du} f1(1) = \frac{d}{du} f2(1) = 0$$

− la dérivée seconde en A ne dépend que de a_1 et a_2 :

$$\frac{d^2}{du^2} f3(0) = 0 \quad \frac{d^2}{du^2} f1(1) = 0$$

18.2.4.2 détermination des polynômes de Bezier

Plaçons nous encore dans le cas de trois points : On a 12 conditions. On peut donc prendre trois fonctions du troisième degré : soit fi $(u)=a.u^3 + b.u^2 + c.u + d$

la dérivée sera donc f'i $(u)=3.a.u^2 + 2.b.u + c$,
et la dérivée seconde f''i $(u)=6.a.u + 2.b$

d'où : f1 (0)=0 => d=0
f''1 (1)=0 => 6a + 2b = 0 d'où b=−3a
f'1 (1)=0 => 3a−6a+c=0 d'où c=3a
f1 (1)=1 => a+b+c = a+3a−3a = 1 d'où a=1, b=−3, c=3

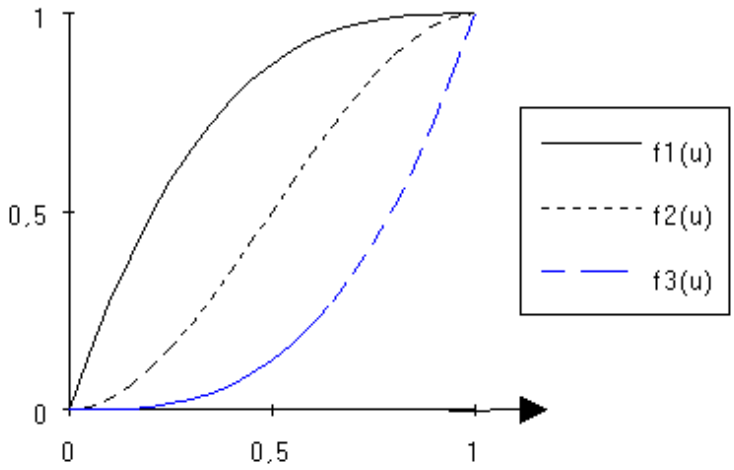
Donc f1(u) = $u^3 - 3.u^2 + 3.u$

de même : f2 (0)=0 => d=0
f'2 (0)=0 => c=0
f2 (1)=1 => a+b=1 d'où b=1−a
f'2 (1)=0 => 3a+2b = 3a+2−2a = 0 d'où a=−2, b=3

donc f2 (u) = $-2\,u^3 + 3.u^2$

et : f3 (0)=0 => d=0
f'3 (0)=0 => c=0
f''3 (1)=0 => b=0
f3 (1)=1 => a=1 donc f3 (u) = u^3

tracé de ces fonctions $f_{im}(u)$ dans le cas m=3 :



P Bezier nous a déterminé le cas général (à l'ordre m) :

$$f_{im}(u) = \frac{(-u)^i}{(i-1)!}\frac{d^{i-1}}{du^{i-1}}\left(\frac{(1-u)^m-1}{u}\right) \quad \text{ou} \quad f_{im}(u) = \sum_{k=i}^{k=m} C_m^k \; C_{k-1}^{i-1} \; (-1)^{i+k} u^k \quad \text{avec} \quad C_a^b = \frac{a!}{b!(a-b)!}$$

Cette seconde écriture se traduit facilement de manière informatique, bien que comportant beaucoup de calculs.

18.2.4.3 forme polynomiale

Plutôt que d'utiliser la formulation vectorielle, on définit le polygone caractéristique par les coordonnées x_i,y_i,z_i de ses sommets. On obtient les coordonnées de tout point de la courbe:

$X(u) = \sum_{i=0}^{m} X_i C_n^i u^i (1-u)^{m-i}$	$Y(u) = \sum_{i=0}^{m} Y_i C_n^i u^i (1-u)^{m-i}$	$Z(u) = \sum_{i=0}^{m} Z_i C_n^i u^i (1-u)^{m-i}$

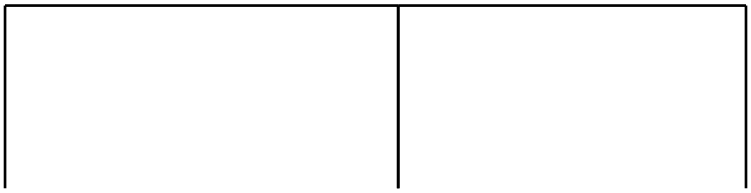
Pour accélérer les calculs, on cherchera à stocker dans des tableaux les valeurs intermédiaires réutilisées plusieurs fois pour un tracé. D'autres formulations ont également été développées, on peut les trouver dans divers ouvrages, par exemple dans la collection "Maths et CAO".

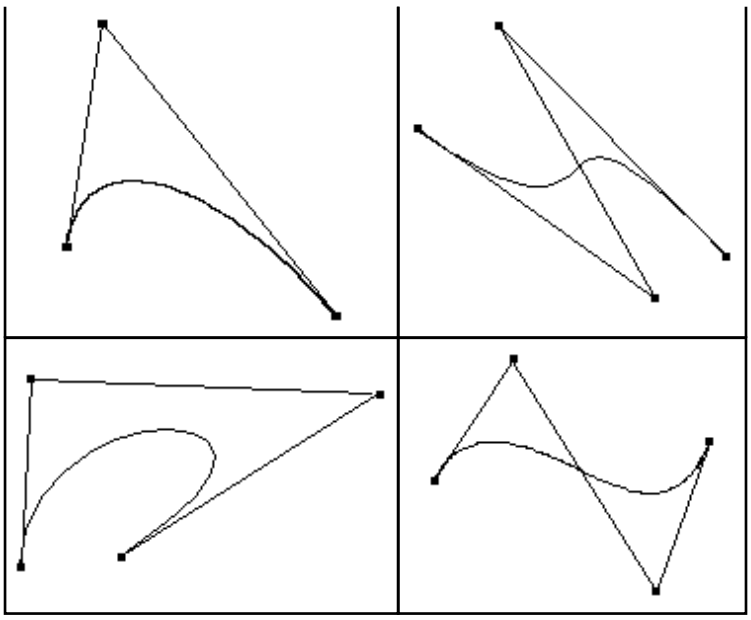
18.2.4.4 intérêt

On peut définir une courbe complexe avec assez peu de points caractéristiques. Certes la courbe ne passe pas par ces points, mais avec un peu d'expérience, on "sent" facilement comment déplacer quelques points caractéristiques pour modifier une courbe. Ceci est encore plus intéressant pour des surfaces tridimensionnelles, où les modifications elles aussi se limiteront aux déplacements de quelques points.

18.2.4.5 exemples

Ces exemples nous montrent la diversité des courbes possibles avec trois ou quatre points :





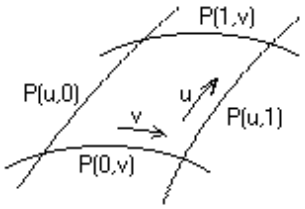
18.3 surfaces

Représenter une surface à l'aide d'un minimum de points est évidement encore plus complexe. De plus, si l'on désire pouvoir modifier facilement une surface, il faut pouvoir le faire en modifiant un minimum de points caractéristiques. On définit en général les surfaces à l'aide de courbes support. Nous nous limitons ici au cas (assez simple) des surfaces de Coons et aborderons les surfaces de Bezier.

18.3.1 Coons

On définit la surface par un réseau de courbes croisées (isoparamétriques). Ceci nous permet de définir des "carreaux", que l'on définit complètement par interpolation surfacique à partir des frontières.

Définissons un carreau et deux paramètres u,v dans [0,1].



la frontière est définie par les quatre équations de courbes suivantes :

−−>	−−>	−−>	−−>
P(u,0)	P(u,1)	P(0,v)	P(1,v)

Déterminons chaque point du carreau par :

$$
\begin{aligned}
\overrightarrow{P(u,v)} = {} & \overrightarrow{P(u,0)}.f_0(v) + \overrightarrow{P(u,1)}.f_1(v) \\
& + \overrightarrow{P(0,v)}.f_0(u) + \overrightarrow{P(1,v)}.f_1(u) \\
& - \overrightarrow{P(0,0)}.f_0(u).f_0(v) - \overrightarrow{P(0,1)}.f_0(u).f_1(v) \\
& - \overrightarrow{P(1,0)}.f_1(u).f_0(v) - \overrightarrow{P(1,1)}.f_1(u).f_1(v)
\end{aligned}
$$

Les conditions nécessaires au passage de la surface par les frontières sont : $f_0(0)=1$, $f_0(1)=0$, $f_1(0)=0$, $f_1(1)=1$. On peut choisir n'importe quelle solution, par exemple $f0(t)=1-t$, $f_1(t)=t$, ce qui permet de définir une surface limitée par les frontières définies. Mais pour que le principe ait une quelconque utilité, il faut au minimum une continuité en tangence entre les carreaux. Coons propose pour cela :

$f_0(t) = 2t^3 - 3t^2 + 1$

$f_1(t) = -2t^3 + 3t^2$

De même, pour obtenir une continuité en courbure, on pourra prendre (à condition évidement d'avoir des courbes isoparametriques continues en courbure) :

$f_0(t) = -6t^5 + 15t^4 - 10t^3 + 1$

$f_1(t) = 6t^5 - 15t^4 + 10t^3$

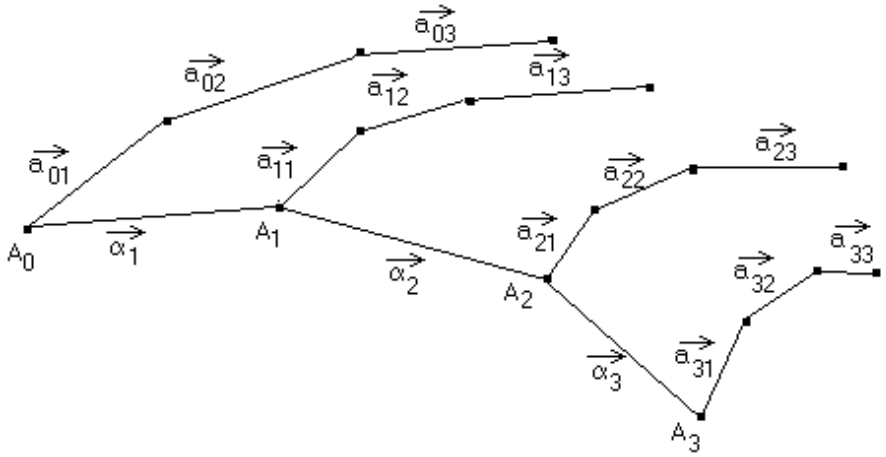
18.3.2 surfaces de Bezier

On utilise un réseau croisé de polygones de Bezier. Toutes les courbes "parallèles" doivent donc être définies par le même nombre de points caractéristiques. Nous supposerons disposer de m+1 courbes d'ordre n, la courbe i étant définie par :

$$\vec{P_i}(u) = \vec{OA_i} + f_{1n}(u)\,\vec{a_{i1}} + \ldots + f_{nn}(u)\,\vec{a_{in}} \quad (i \text{ dans } [0,m])$$

Les points A_i forment eux aussi un polygone caractéristique. On note a_i le vecteur $A_{i-1}A_i$.

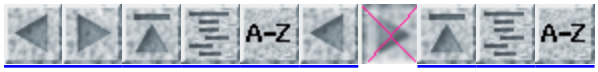
Les points caractéristiques de même "niveau" forment des polygones caractéristiques "perpendiculaires". Mais les courbes définies par ces polygones ne se croisent pas nécessairement (puisqu'elles ne passent pas par les points caractéristiques). La surface elle non plus ne passe pas par ces courbes, mais elle est bordée par les quatre courbes frontières.



Un point de la surface sera défini par deux paramètres (u et v)

$$\vec{P}(u,v) = \vec{OA_0} + \sum_{i=1}^{m} \vec{a_i}\, f_{im}(u) + \sum_{j=1}^{n} \vec{a_{0j}}\, f_{jn}(v) + \sum_{i=1}^{m}\sum_{j=1}^{n} (\vec{a_{ij}} - \vec{a_{i-1,j}})\, f_{im}(u)\, f_{jn}(v)$$

Je ne serai pas plus précis ici, de nombreux ouvrages traitant de ce problème, en particulier le livre de P. Bezier, dans la collection Maths et C.A.O. chez Hermes.



19 ANNEXE : DOCUMENTATION DE LA BIBLIOTHEQUE GRAPHIQUE

19.1 G2D Librairie graphique de base en C

19.1.1 Préliminaire

Cette bibliothèque permet de dessiner sur tout écran graphique, et ce sur tout matériel, moyennant peu de modifications. Cette bibliothèque a été écrite de manière à être le plus facilement portable que possible. On peut utiliser au maximum le C standard (ANSI), les fonctions de base faisant des appels au BIOS (passer en mode graphique, allumer un point). Mais pour accélérer les affichages, on utilise les fonctions de l'unité GRAPH de Turbo C, les parties programmées de manière plus standard restant dans le source de la bibliothèque, en commentaires. Pour l'utilisateur, il n'y a pas d'autre différence que la rapidité, les fonctions disponibles dans les deux cas sont les mêmes.

Il faut avant tout définir, dans votre programme, l'ensemble des fonctions que vous allez utiliser. Ceci se fait par un fichier inclus, en C **#include "G2D.H"** (entre doubles quotes pour chercher le fichier dans le répertoire actuel). De plus, il faudra lier la bibliothèque à votre programme lors de l'édition de liens : LINK MONPROG.OBJ+G2D.OBJ MONPROG.EXE pour l'éditeur de liens MS−DOS. Sous TURBO−C il faut créer un projet (Project −> New Project) puis en y insérant tous les modules nécessaires (Project −> Add Item, insérer G2D.C, VOTREPROG.C, ainsi que les autres bibliothèques nécessaires).

Toutes les fonctions de cette bibliothèque commencent par la lettre G, afin de les identifier plus facilement.

19.1.2 Passage en mode graphique

Tout dessin doit débuter par un passage en mode graphique par la fonction **void ginit(gtypecr type_écran)**. GINIT efface l'écran. Dans le cas d'un PC, type_écran peut valoir :

0 : mode texte. On est limité à 25 lignes de 80 caractères, sans graphique possible.

1 : CGA couleur (320x200 en 4 couleurs)
2 : CGA mono (640x200)
3 : écran Olivetti (640x400 mono)
4 : EGA (640x350 16 couleurs)
5 : MDA (320x200 256 couleurs)
6 : VGA (640x480 16 couleurs)
7 : SVGA 640x480 256 couleurs
8 : SVGA 800x600 16 couleurs
9 : Hercules 720x348 monochrome

GINIT crée la fenêtre 0 (réserve tout l'écran au graphique). 0n peut utiliser la fonction **gtypecr gtyp_ecr_par_def(void)** qui rend une valeur par défaut dépendant de votre matériel (en général 6 ou 2). De plus la valeur rendue par cette fonction peut être forcée sous MS DOS lors de l'appel de votre programme compilé par des options /CGA, /EGA, /VGA , /SVGA, /HER ou /OLI. Dans le cas d'une version utilisant les spécificités de TURBO C, il vous faudra fournir le fichier .BGI correspondant à votre écran soit dans le répertoire actuel, soit dans un répertoire défini par SET BGI=répertoire (par exemple, mettez dans AUTOEXEC.BAT la commande SET BGI=C:\TC\BGI).

On quitte le mode graphique et retourne en mode texte par la fonction **void gfin(void)**. Il ne faut jamais ré−appeler GINIT sans être passé par GFIN (si ce n'était que pour effacer l'écran il vaut mieux utiliser **gefface** dans la fenêtre 0).

19.1.3 Les fenêtres

Une fois en mode graphique, on peut définir des fenêtres. Ceci n'est pas obligatoire, on sera sans cela dans la fenêtre 0 (totalité de l'écran). On crée une fenêtre par :

void gcree_fenetre(float xgauche, float xdroit, float ybas, float yhaut, int num_fen). On y définit les

limites de la fenêtre par rapport à l'écran complet, par des valeurs entre 0 (à gauche en X et en bas en Y) et 1 (à droite en X, en bas en Y). Num_fen est le numéro que l'on désire attribuer à la fenêtre, il doit être un entier strictement positif, mais on n'est pas obligé de les faire se suivre. Une fenêtre est une partie de l'écran, pour laquelle on s'est défini une échelle, une couleur de tracés, une taille de caractères... Mais cette fenêtre n'est qu'une limite logique (on ne dépassera pas ses limites), sa création ne changeant rien à l'écran (on peut utiliser GEFFACE et GCADRE). On peut se faire superposer des fenêtres avec des échelles différentes, les tracés se superposeront. On appellera une telle fenêtre : "fenêtre écran", ou "fenêtre papier" sur table traçante.

Si l'on a défini plusieurs fenêtres, on passe de l'une à l'autre par **void gchoix_fenetre(int num_fen)**. On dessine toujours dans la dernière fenêtre choisie. GCREE_FENETRE nous met directement dans la nouvelle fenêtre créée. Dans chaque fenêtre sont mémorisés : les échelles (2D et 3D le cas échéant), les couleurs, le type de trait et les informations sur les écritures (taille, direction, position).

On supprime une fenêtre par **void gferme_fenetre(int num_fen)**. La fermeture d'une fenêtre ne change rien à l'écran, elle ne fait qu'oublier les définitions dans cette fenêtre. Il vaut mieux ne pas être dans la fenêtre à fermer, on se retrouvera dans ce cas dans la fenêtre 0.

Tout dessin sera limité aux limites de la fenêtre actuelle. Des algorithmes ont été mis en place pour accélérer le traitement des dépassements de la fenêtre (clipping / clôture). Vous pouvez donc sans soucis tracer l'ensemble de votre dessin, même si seule une partie entre dans la fenêtre, la perte de temps restera assez faible.

On efface la fenêtre actuelle par **void geffface(void)** (du moins on remplit la fenêtre par la couleur du fond). On encadre la fenêtre actuelle par **void gcadre(void)**. Pour effacer la totalité de l'écran, utiliser GEFFACE après avoir choisi la fenêtre 0.

la fonction **int gexiste_fenetre(int num_fen)** permet de déterminer si une fenêtre est encore ouverte (1) ou non (0).

19.1.4 Les échelles

Vous travaillez toujours à votre échelle, avec des coordonnées de type **gcoord** (actuellement float). On appelle "fenêtre utilisateur" la partie de l'espace (dans votre système de coordonnées) que vous désirez représenter dans la "fenêtre écran". On définit l'échelle ("coordonnées utilisateur") par la fonction **void gechelle(gcoord xgauche, gcoord xdroite, gcoord ybas, gcoord yhaut)**. Vous définissez la coordonnée en X que vous désirez donner au bord gauche de la fenêtre écran, puis le bord droit, puis vous définissez les bord haut et bas. Les valeurs n'ont pas besoin d'être croissantes en X ou en Y, ni de commencer à 0, ni d'être du même ordre de grandeur en X et Y (vous pouvez représenter en X des jours (entre 0 et 31), et des kilomètres de saucisse en Y (entre 0 et 0.1). Dans ce cas, la fonction GCERCLE vous dessinera une ellipse à l'écran.

Si vous désirez une unité représentée par la même distance en X et en Y, utilisez **void gechelleort(gcoord xgauche, gcoord xdroite, gcoord ybas, gcoord yhaut)**. Quelles que soient les proportions de la fenêtre écran actuelle et de la fenêtre utilisateur désirée, cette fonction choisira une échelle représentant la fenêtre utilisateur la plus grande possible, à l'intérieur de la fenêtre écran, tout en gardant un repère orthonormé. Le clipping se fera sur la fenêtre écran, vous pourrez donc dessiner un partie hors de la fenêtre utilisateur : si la fenêtre écran est deux fois plus longue que haute, gechelleort(0, 100, 0, 100) permettra de dessiner de 0 à 100 en Y, mais de −50 à 150 en X. GCERCLE dessinera un véritable cercle. La fonction **void glimites(gcoord *xgauche, gcoord *xdroite, gcoord *ybas, gcoord *yhaut)** permet de récupérer les vraies limites de la fenêtre (dans votre échelle utilisateur).

19.1.5 Attributs de tracé

On peut définir, dans chaque fenêtre, une couleur de tracé par la fonction **void gcouleur(int trait)**. Les tracés qui suivront se feront dans la couleur définie. La couleur peut être choisie entre 0 (couleur du fond, en général

le noir) et la variable entière **gcoul_max** (qui dépend de votre type d'écran). Si trait est supérieur à gcoul_max, c'est gcoul_max qui sera choisi, ce qui signifie que si vous effectuez, sur un écran monochrome ou une imprimante, un dessin initialement prévu en couleur, tout ce qui était prévu dans une autre couleur que celle du fond (0) sera dessiné (couleur 1).

On peut également définir une couleur spécifique pour les remplissages par **void gcouleur_remplissage(int rempli)**, ainsi que la couleur du fond par **void gcouleur_fond(int fond)**. **void gcouleur_tot(int trait, int rempli, int fond)** permet de définir en une fois les trois couleurs (utilisé par la bibliothèque de remplissage).

Sur un écran couleur le permettant, vous pouvez redéfinir les couleurs de base, par la fonction **void gpalette(int num_coul, float rouge, float vert, float bleu)**. Rouge, vert, bleu sont des réels entre 0 et 1 définissant l'intensité de chaque couleur de base (0 éteint, 1 intensité maximale). **void gpalette_progressive(void)** crée une palette de couleurs progressives de 1 à gcoul_max−1, du bleu au rouge, avec 0 (couleur du fond) en noir et gcoul_max en blanc (pour les tracés). On retourne à la palette par défaut du système par **void gpalette_initiale(void)** (sur PC, ne pas l'oublier avant le GFIN, sinon vos couleurs sous DOS peuvent être modifiées).

Certaines versions de G2D permettent de définir le type de trait par **void gtypetrait(int typ)** (0 trait continu, 1 interrompu, 2 mixte, la taille est définie par **gtaillecar** (voir documentation GECR). De même **void geptrait(int ep)** permet de définir l'épaisseur du trait. Le test de ces options ralentissant beaucoup le tracé, ils n'ont pas été implantés dans toutes les versions.

19.1.6 Tracés de base

Tous les tracés se font dans la couleur définie par le dernier GCOULEUR appelé dans cette fenêtre. On peut effacer un objet en le dessinant en couleur 0. Toutes les coordonnées seront de type **gcoord** (réel) dans votre système de coordonnées (coordonnées utilisateur), défini par le dernier GECHELLE appelé dans cette fenêtre.

La fonction **void gpoint(gcoord x, gcoord y)** allume le pixel le plus proche du point réel désiré, la fonction **int gtestpoint(gcoord x, gcoord y)** donne la couleur actuelle d'un point, la fonction **void gcarpoint(gcoord x, gcoord y, int typ)** trace en un point un signe (0=point, 1=X, 2=carré, 3=*, 4=losange, 5=+). Cette fonction est utile pour les tracés de courbes.La taille est définie par GTAILLECAR (voir documentation GECR).

La fonction **void gligne(gcoord x1, gcoord y1, gcoord x2, gcoord y2)** trace une ligne du point 1 au point 2, **void gfleche(gcoord x0, gcoord y0, gcoord x1, gcoord y1)** trace une flèche (signe flèche sur le point 2). **void grectangle(gcoord xg, gcoord yb, gcoord xd, gcoord yh)** trace un rectangle parallèle aux axes.

La fonction **void garcellipse(gcoord xcentre, gcoord ycentre, gcoord rayonx, gcoord rayony,gcoord angledeb, gcoord anglefin)** trace un arc d'ellipse défini par son centre, ses rayons en X et Y, l'angle de début et l'angle de fin (sens trigonométrique, en degrés). On peut également utiliser les fonctions **void garc(float xcentre, float ycentre, float rayonx, float angledeb, float anglefin)**, **void gcercle(float xcentre, float ycentre, float rayon)** et **void gellipse(float xcentre, float ycentre, float rayonx, float rayony)**.

19.1.7 Table traçante

On peut faire tous les tracés sur table traçante ou imprimante graphique, en même temps que l'on dessine sur l'écran. Tous les dessins se feront à la précision de la table traçante, même si celle−ci est différente de la définition de l'écran (en général elle est meilleure). Le tracé débutera à l'appel de **void gdebut_table(gtypecrn, char *nomfic,float xgauche, float xdroite, float ybas, float yhaut)**. Typ représente le type de table (1 et 2 table Houston DMP 40 A4 et A3, 3 et 4 imprimante graphique matricielle type EPSON LX en mode paysage ou portrait, nécessitant beaucoup de mémoire, 5 et 6 EPSON LX utilisant peu de mémoire mais des accès disques nombreux, ce qui rend l'impression très lente), 7 et 8 table traçante HPGL (HP7470 par ex), en mode paysage ou portrait. Nomfic représente le nom du fichier ASCII comportant les commandes pour la table, qu'il faudra imprimer par la suite (sous MS DOS, on peut choisir PRN ou COM1 pour un envoi

immédiat), mais il vaut mieux utiliser COPY /B NOMFIC PRN (copie en format binaire). Xgauche, Xdroite, Ybas, Yhaut sont des réels entre 0 et 1 qui définiront la portion de la feuille sur laquelle on "collera" la totalité de l'écran graphique (ceci permet de réduire le dessin, voire de le retourner). A partir de ce moment, il va falloir définir toutes les fenêtres et échelles, puis dessiner. Les effacements ne fonctionneront pas (sauf sur imprimante matricielle). Ce n'est qu'en appelant la fonction **void gfin_table(void)** que le fichier sera correctement fermé et que l'impression pourra débuter. On pourra ensuite soit continuer à dessiner sur l'écran, soit quitter le graphique par GFIN. Pour un résultat similaire sur l'écran et sur la table traçante choisissez un mode paysage.

19.1.8 Autres bibliothèques

Des fonctions supplémentaires ont été regroupées dans des bibliothèques différentes de G2D, afin de ne charger que celles utiles pour votre programme (voir détails plus bas) :
GECR : écritures graphiques
GPOLY : traçage, hachurage, remplissage de polygones
GREMPL : remplissage d'une surface déjà dessinée à l'écran
G3D : bibliothèque tridimensionnelle (ainsi que G3POLY et G3ECR)
SOURIS : prise en compte de la souris, directement dans votre échelle utilisateur
MENUG : menus graphiques
etc...

19.1.9 Exemple :

```
#include <conio.h>
#include <stdlib.h>
#include "g2d.h"
#define alea ((float)rand()/RAND_MAX)

void main(void)
{
 int i;

 ginit(gtyp_ecr_par_def());
 gefface();
 gcadre();
 gcree_fenetre(0,0.2,0.2,0.5,20);
 gcadre();
 gechelle(0,1,0,1);
 for (i=0;i<1000;i++)
  {
   gcouleur(rand()%16);
   gpoint(alea,alea);
  }
 gcree_fenetre(0.2,0.4,0.2,0.5,21);
 gcadre();
 gechelle(0,1,0,1);
 gcercle(0.5,0.5,0.5);
 gtaillecar(0.1,0.1);
 for (i=0;i<19;i++)
  {
   gcouleur(rand()%16);
   gcarpoint(alea,alea,rand()%6);
  }
 gcree_fenetre(0,0.2,0.5,0.8,22);
 gcadre();
 gechelle(0,1,0,1);
 for (i=0;i<100;i++)
  {
   gcouleur(rand()%16);
   gligne(alea,alea,alea,alea);
```

```
  }
 gcree_fenetre(0.2,0.4,0.5,0.8,24);
 gcadre();
 gechelle(0,1,0,1);
 for (i=0;i<20;i++)
  {
   gcouleur(rand()%16);
   grectangle(alea,alea,alea,alea);
  }
 gcree_fenetre(0.5,0.9,0.1,0.9,23);
 gcadre();
 gechelleort(0,1,0,1);
 g
taillecar(0.05,0.05);
 for (i=0;i<30;i++)
  {
   gcouleur(rand()%16);
   gfleche(alea,alea,alea,alea);
  }
 getch();
 gfin();
}
```

19.2 bibliothèque GECR

Cette bibliothèque, associée à G2D, permet d'afficher du texte en graphique. Son intérêt sur écran est moindre (on pouvait utiliser printf), bien que l'on puisse définir la position du texte de manière précise, à n'importe quelle taille et dans n'importe quelle direction. Mais les caractères obtenus sont moins beaux et moins lisibles que par printf. Mais par contre ils permettent d'écrire sur n'importe quel organe de sortie, à condition de savoir initialiser cet organe et y tracer une droite (table traçante, voire fraiseuse). Les caractères ne sont vraiment lisibles que pour une taille supérieure à 6 pixels. Certaines fonctions de traitement de chaînes de caractères, ainsi que le type **chaine** sont définis dans la petite bibliothèque CHAINES.

Avant tout, il faut déclarer dans votre programme les fonctions de la bibliothèque (**#include "G2D.H" #include "GECR.H"**). Il faut ensuite initialiser le graphique, définir vos fenêtres (avec leur échelle respective). Puis vous pouvez utiliser :

void gecrit(gcoord x, gcoord y, chaine gtexte) : écrit votre chaine de caractères en commençant en x,y,

void gecrite(gcoord x, gcoord y, int i) : idem pour un entier,.

void gecritr(gcoord x, gcoord y, float r) : idem pour un réel

void gtaillecar(gcoord x, gcoord y) : définit la taille des caractères, dans votre échelle, en X et Y. Si vous choisissez une échelle en Y de haut en bas (pour simuler des numéros de ligne d'écran tels qu'ils sont couramment définis), il faut donner une taille de caractères négative en Y pour ne pas écrire à l'envers.

void gdircar(float d) : pour donner la direction d'écriture, en degrés.

Si vous ne vous souvenez pas de votre échelle réelle (gechelleort ou g3d), vous pouvez utiliser **void gtaillecarpc(float pcx, float pcy)**. pcx et pcy sont des réels entre 0 et 1, et définissent la taille des caractères en proportion de la taille de la fenêtre.

Remarque : gtaillecar et gtaillecarpc servant également à définir la taille des carpoint et gfleche, ils sont définis dans G2D.

Les fonctions **gcoord gcurseurx(void)** et **gcoord gcurseury(void)** rendent la position du "curseur", c'est à dire la position où il aurait écrit le prochain caractère. Ceci permet de mettre côte à côte des chaînes, entiers

et réels.

Les écritures se SUPERPOSENT au dessin déjà présent à l'écran, pour effacer une ligne de texte avant d'en réécrire une autre, on peut : soit réécrire l'ancienne chaîne en couleur de fond, soit appeler **void gecrit_blanc(gcoord x, gcoord y, int nbcar)**, définie dans la bibliothèque GPOLY, qui remplit en couleur de fond un rectangle en X,Y, de longueur nbcar caractères et de hauteur 1 caractère (prend en compte votre taillecar et dircar).

Cette bibliothèque est assez lente, et prend pas mal de place en mémoire. Elle est définie dans GECR_STD.C, qu'il faudra lier à votre projet (ou LINKer). En cas d'une utilisation exclusivement limitée à l'écran, la bibliothèque définie dans GECR_ECR.C permet, par appel aux fonction spécifiques du compilateur, une écriture plus rapide. Les fonctions ayant le même nom, il est inutile de changer l'inclusion de GECR.H dans votre source. Cependant les caractères seront dessinés à taille constante (8 pixels sous TurboC) et direction constante. Par contre, chaque caractère d'une chaîne sera dessiné à la même position qu'avec la bibliothèque standard (donc en fonction gtaillecar et gdircar). De plus cette version rapide ne sait pas écrire sur les tables traçantes (et imprimantes).

EXEMPLE GECR : il est intéressant de linker ce test avec GECR_STD en premier, puis avec GECR_ECR pour comparer.

```
#include <conio.h>
#include "g2d.h"
#include "gecr.h"

void main(void)
 {
  int i,j;
  ginit(gtyp_ecr_par_def());
  gefface();
  gcadre();

/*essai de différentes tailles*/
  gechelle(0,639,0,479);
  gdircar(0);
  gcouleur(1);
  j=2;
  gtaillecar(j,j);
  gecrit(0,475,"\n");
  for (i=1;i<=17;i++)
   {
    j+=2;
    gtaillecar(j,j);
    gecrit(10,gcurseury(),"taille ");
    gecrite(gcurseurx(),gcurseury(),j);
    gecrit(gcurseurx(),gcurseury(),"\n");
   }

/*essai de directions et rapport X/Y différent*/
  gechelle(0,1,0,1);
  gtaillecar(0.02,0.03);
  gcouleur(10);
  gecrit(0.1,0.5,"salut");
  gtaillecar(0.02,0.18);
  gcouleur(2);
  gecrit(0.7,0.13,"HAUT");
  gtaillecar(0.2,0.05);
  gcouleur(3);
  gecrit(0.05,0.05,"LARGE");
  gtaillecar(0.03,0.04);
  gdircar(15);
```

```
  gcouleur(4);
  gecrit(0.5,0.34,"ça monte !");
  gdircar(90);
  gcouleur(5);
  gecrit(0.9,0.15,"ça grimpe.");

/*fin de la démo*/
  getch();
  gfin();
 }
```

19.3 remplissage − hachurage de polygones (GPOLY)

la bibliothèque **GPOLY** permet de faire des remplissages ou hachurages indépendamment de ce qui se trouve déjà à l'écran. La frontière n'est pas dessinée lors d'un remplissage, c'est à vous d'appeler **gpolygone** pour la tracer si nécessaire. Le polygone à tracer est défini par ses N sommets, définis par deux tableaux de type **gtableau_polygone** (un pour chaque coordonnée). Ce sont des tableaux de réels limités à une taille de **GMAXPOLY** = 10 éléments. Par contre, vous pouvez redéfinir GMAXPOLY par un #define avant l'inclusion de GPOLY.H. Ceci vous permet de déclarer des variables de type **gtableau_polygone** à la dimension désirée. Mais il vous est aussi possible de déclarer directement des tableaux de **gcoord** (réels) et les transmettre en arguments aux fonctions de GPOLY. En effet, les fonctions ne font qu'utiliser votre tableau, sans en créer d'autre (ou alors par création par malloc).

void gpolygone(gtableau_polygone x, gtableau_polygone y, int nombre) : trace un polygone en reliant les points (x,y tableaux de reels, nombre : nombre de sommets du polygone). Cette fonction ne tracant que des segments de droites, les polygones peuvent être quelconques (mais fermés).

void gremplit_polygone(gtableau_polygone x, gtableau_polygone y, int nb) remplit le polygone dans la couleur de remplissage donnée (définie par la fonction **gcouleur_remplissage** de G2D). Dans la version standard (mais plus lente et encore plus gourmande en mémoire) GPOLY_STD, si votre écran est monochrome (variable **gcoul_max** de G2D valant 1), vous obtiendrez des niveaux de gris par répartition aléatoire de points. Cette fonction ne fonctionne bien que sur écran, utilisez le hachurage sur table traçante (avec interhachures de l'épaisseur du trait au minimum).

void ghachure_polygone(gtableau_polygone x, gtableau_polygone y, int nb, float angle,float interligne) : Cette fonction permet de hachurer un polygone, en donnant l'angle des hachures et la valeur de l'interligne La couleur des hachures est celle des remplissages (définie par **gcouleur_remplissage**).

Remarque : on trouve également dans cette unité la fonction **gecritblanc(gcoord x, gcoord y, int nbcar),** qui remplit un rectangle en couleur du fond, ce qui permet d'effacer un texte écrit par GECR (tient compte de la taille et direction d'écriture)

Exemple : (avec ghachure_polygone, on peut essayer gremplit_polygone)

```
#include <conio.h>
#include "g2d.h"
#include "gpoly.h"
#include "gecr.h"

void main(void)
 {
  gtableau_polygone x,y;
  int i;
  ginit(gtyp_ecr_par_def());
  gcree_fenetre(0,1,0.5,1,1);
  gcouleur(14);
  gcadre();
  gechelle(0,4,0,4);
```

```
  gecrit (0.1,0.1,"polygones convexe");
  x[0]=1;y[0]=1;
  x[1]=1;y[1]=2;
  x[2]=1.6;y[2]=3.8;
  x[3]=2.4;y[3]=3.8;
  x[4]=3;y[4]=2;
  x[5]=3;y[5]=1;
  for (i=0;i<=12;i++)
   {
    gcouleur_remplissage(i+1);
    ghachure_polygone(x,y,6,(3.14/15)*i,0.1);
   }
  gpolygone(x,y,6);

  gcree_fenetre(0,0.5,0,0.5,2);
  gcadre();
  gechelle(0,4,0,4);
  gecrit (0.1,0.1,"polygone NON convexe");
  x[0]=1;y[0]=1;
  x[1]=2;y[1]=3.5;
  x[2]=3;y[2]=1;
  x[3]=2;y[3]=2.5;
  for (i=0;i<=12 ;i++)
   {
    gcouleur_remplissage(i+1);
    ghachure_polygone(x,y,4,(3.14/15)*i,0.1);
   }
  gpolygone(x,y,4);

  gcree_fenetre(0.5,1,0,0.5,3);
  gcadre();
  gechelle(0,4,0,4);
  gecrit (0.1,0.1,"faux polygone");
  x[0]=1;y[0]=1;
  x[1]=2;y[1]=3;
  x[2]=3;y[2]=1;
  x[3]=1;y[3]=2.4;
  x[4]=3;y[4]=2.4;
  for (i=0;i<=12;i++)
   {
    gcouleur_remplissage(i+1);
    ghachure_polygone(x,y,5,(3.14/15)*i,0.1);
   }
  gpolygone(x,y,5);
  getch();
  gfin();
 }
```

19.4 remplissage suivant l'écran (GREMPL)

On peut également utiliser la bibliothèque GREMPL qui permet de remplir (d'une couleur de remplissage donnée) une frontière déjà dessinée à l'écran. Il suffit de donner un point de départ par ses coordonnées X et Y, la fonction **void gremplir(gcoord x, gcoord y)** remplira autour de ce point jusqu'à trouver des points déjà allumés. Il est évidement impossible de **recouvir** de cette manière des objets déjà dessinés. La frontière doit être dessinée auparavant, mais peut avoir n'importe quelle forme (voir plus bas l'exemple d'une spirale). Si la frontière n'est pas fermée, on remplira jusqu'aux limites de la fenêtre (même si le cadre n'a pas été tracé). Si le point de départ correspond à un point déjà allumé (par exemple sur une frontière), rien n'est rempli. En version spécifique Turbo, ne s'arrête que sur la couleur actuelle de tracé, en version standard s'arrête sur toute couleur différente du fond.

exemple :

```
#include <stdio.h>
#include <conio.h>
#include <math.h>

#include "g2d.h"
#include "gecr.h"
#include "grempl.h"

void figure1(void)
 {
  gligne(90,10,600,10);
  gligne(90,10,90,350);
  gligne(90,350,250,200);
  gligne(250,200,600,350);
  gligne(600,350,600,10);
 }

void figure3(void)
 {
  int a;
  for (a=1;a<=15;a++)
   {
    gcouleur(a);
    gligne(a*40,70,(a+1)*35,70);
   }
  grectangle(100,100,300,300);
  gcadre();
 }

void figure2(void)
 {
  int r, x1, y1, x2, y2,xlimite,ylimite;
  float o;
  xlimite=640;ylimite=480;
  o= 0;
  x1= xlimite / 2;
  y1= ylimite / 2;
  for (r=0;r<=200;r++)
     {
         x2= (int)(sin(o)*r*(xlimite/640.0))+xlimite / 2;
         y2= (int)(cos(o)*r*(ylimite/480.0))+ylimite / 2;
         gligne(x1,y1,x2,y2);
         o= o+PI/18.0;
         x1= x2;
         y1= y2;
     }
  gligne(x1, y1, x1, y1+ylimite/10);
 }

void figure4(void)
 {
  grectangle(40,40,600,440);
  gtaillecar(55,200);
  gecrit(60,100,"C'est Fini");
 }

void main(void)
 {
   ginit(gtyp_ecr_par_def());
   gcree_fenetre(0,0.5,0,0.5,1);
   gechelle(0,640,380,−20);
   gcadre();
   gcouleur(13);
   figure1();
   gcouleur_remplissage(3);
```

```
   gremplir(100,300);

   gcree_fenetre(0.5,1,0,0.5,2);
   gechelle(0,640,0,480);
   gcouleur(14);
   gcadre();
   figure2();
   gcouleur_remplissage(2);
   gremplir(300,300);

   gcree_fenetre(0,0.5,0.5,1,3);
   gechelle(0,640,0,480);
   figure3();
   gcouleur_remplissage(1);
   gremplir(400,300);

   gcree_fenetre(0.5,1,0.5,1,4);
   gcadre();
   gechelle(0,640,0,480);
   gcouleur(13);
   figure4();
   gcouleur_remplissage(4);
   gremplir(300,300);

   puts("c'est fini");
   getch();
   gfin();
}
```

19.5 bibliothèque GPLUS

Cette bibliothèque contient quelques petites fonctions qui peuvent être utiles :

void gix(gcoord x, gcoord y) dessine une petite croix (taille 4 pixels) autour du point X,Y en inversant la couleur des pixels, ce qui fait qu'un second appel remet l'écran tel qu'il se trouvait auparavant. Cette fonction est utilisée par MENUG pour représenter le curseur graphique lorsque la souris est absente.

void greticule(gcoord x, gcoord y) : idem GIX mais dessine une croix de la taille de la fenêtre. C'est évidement plus lent, mais plus visible.

void gcadre_ombre(int coul_fond) trace un fond de fenêtre avec effet de relief (comme les boutons windows)

19.6 bibliothèque tridimensionnelle

19.6.1 G3D

Cette bibliothèque, associée à G2D, permet de tracer des primitives tridimensionnelles, à votre échelle. Vous donnez tous vos points par leur 3 coordonnées. Cette bibliothèque vous fera vos projections, avec perspective. Mais cette bibliothèque n'est qu'une aide au tracé de traits, et ne sait pas en retrouver une définition de volume. Elle ne pourra donc pas éliminer des faces cachées, puisqu'elle ne connaît pas les faces. Toutes les fonctions de cette bibliothèque commencent par G3, afin de les identifier facilement.

Avant tout, il faut déclarer dans votre programme les fonctions de la bibliothèque (inclure entre autres **G2D.H** et **G3D.H**). Il faut ensuite initialiser le graphique, définir vos fenêtres, les couleurs... Toutes les fonctions 2D seront encore utilisables, la bibliothèque G3D ne faisant que des projections 3D−2D, puis utilise les fonctions de G2D pour tracer.

Dans une fenêtre que vous destinez au 3D, vous devez définir votre projection, c'est à dire la position de l'oeil O de l'observateur (ou sa caméra), ainsi que le point que vous visez V, par la fonction **void g3origine(gcoord gxo, gcoord gyo, gcoord gzo, gcoord gxv, gcoord gyv, gcoord gzv, int propose_axe, int type_3d)**. Si propose_axe est faux (0), l'ordinateur cherchera à mettre l'axe Z vertical sur l'écran, sinon il posera la question. type_3d peut valoir **G3PARAL, G3PERSP** ou **G34VUES**, permettant une projection parallèle, en perspective ou en mode 4 vues (voir plus loin).

Puis vous pouvez définir votre "parallélépipède limite utilisateur" en donnant les minis−maxis que vous désirez faire entrer dans votre fenêtre, dans les 3 directions. Ceci définit votre échelle utilisateur. Pour cela appelez la fonction **void g3echelle(gcoord xmin, gcoord ymin, gcoord zmin, gcoord xmax, gcoord ymax, gcoord zmax)**, mais toujours APRES avoir appelé G3ORIGINE. Si vous désirez changer d'origine pour une même échelle, il faudra appeler à nouveau g3echelle après votre g3origine, ou appeler **void g3change_origine(gcoord gxo, gcoord gyo, gcoord gzo, gcoord gxv, gcoord gyv, gcoord gzv)**. qui recalcule l'échelle (comme dans l'exemple ci−après, même en éloignant l'oeil l'objet restera dessiné en pleine fenêtre).

On peut ensuite effectuer les tracés par les fonctions :

void g3point(gcoord xm, gcoord ym, gcoord zm) affichage d'un point,

void g3ligne(gcoord xm, gcoord ym, gcoord zm, gcoord xn, gcoord yn, gcoord zn) : tracé d'un segment,

void g3fleche(gcoord xm, gcoord ym, gcoord zm, gcoord xn, gcoord yn, gcoord zn) : tracé d'une flèche (dirigée de M vers N),

void g3cercle(gcoord xcentre, gcoord ycentre, gcoord zcentre, gcoord x1, gcoord y1, gcoord z1, gcoord x2, gcoord y2, gcoord z2) : cercle. on donne le centre et 2 points de passage (le 1er donne le rayon, le 2è définit uniquement le plan contenant le cercle). Il ne faut pas que ces 3 points soient allignés (infinité de solutions). Ce cercle sera représenté par une ellipse, sauf dans le cas de projections particulières.

void g3arc(gcoord xcentre, gcoord ycentre, gcoord zcentre, gcoord x1, gcoord y1, gcoord z1, gcoord x2, gcoord y2, gcoord z2, gcoord xpas, gcoord ypas, gcoord zpas) : on donne le centre, les 2 extrémités et un point de passage

On quitte le 3D soit en refermant la fenêtre, soit en y définissant une nouvelle échelle 2D, soit en quittant complètement le graphique par **GFIN.**

19.6.2 précisions sur le mode G34VUES

En définissant par g3origine une fenêtre de type G34VUES, il y a création automatique de 4 sous fenêtres de la fenêtre actuelle. Leur numéros seront définis automatiquementde numfen+ à numfen+4 (numfen étant le ndeg. de la fenêtre actuelle). Ces numéros de fenêtres ne doivent pas correspondre à des fenêtres déjà ouvertes, sous peîne d'erreur. La fenêtre est divisée en 4 parties égales, trois fenêtres en projection parallèle suivant les 3 axes, la quatrième étant une perspective définie par votre g3origine. En appelant une fonction de dessin G3D dans la fenêtre de base, on génèrera automatiquement un tracé dans les 4 sous fenêtres.. Par contre le dessin limité à une seule des sous fenêtres reste possible par un gchoix_fenêtre.

Pour fermer la fenêtre de base, il faut d'abord (en étant dans cette fenêtre) appeler **void g3ferme_sous_fenetres(void)** pour supprimer les 4 sous fenêtres puis faire un gferme_fenetre de la fenêtre de base. Pour changer l'origine, appeler g3change_origine qui contrairement à g3origine ne recréera pas de sous fenêtres.

Ce mode G34VUES n'est pas encore définitif, il devrait encore être amélioré.

19.6.3 G3ECR

On peut aussi écrire des textes, (avec la bibliothèque G3ECR, utilisant GECR), mais on ne donne en 3D que le point de départ, l'écriture se faisant toujours dans le plan de l'écran. Fonctions déclarées dans G3ECR.H : **void g3ecrit(gcoord ggx, gcoord ggy, gcoord ggz, chaine gtexte), void g3ecrite(gcoord ggx, gcoord ggy, gcoord ggz, int ent), void g3ecritr(gcoord ggx,gcoord ggy,gcoord ggz,gcoord r)**. De plus on trouve dans cette bibliothèque la fonction **void g3axes(gcoord xm, gcoord ym, gcoord zm)** qui trace les 3 axes X,Y,Z. Il faut donner les longueurs des axes à cette fonction.

Pour utiliser ces fonctions il faut inclure GECR.H et G3ECR.H.

19.6.4 G3POLY

En incluant G3POLY.H, on peut accéder aux versions 3D de GPOLY. Les polygones doivent être plans (ou presque) :

void g3polygone(gtableau_polygone x, gtableau_polygone y, gtableau_polygone z, int nombre) : trace un polygone en reliant les points (x,y,z tableaux de reels),

void g3remplit_polygone(gtableau_polygone x, gtableau_polygone y, gtableau_polygone z, int nombre) : remplit le polygone dans la couleur de remplissage donnée, si l'écran est monochrome, met des niveaux de gris par répartition aléatoire de points.

void g3hachure_polygone(gtableau_polygone x, gtableau_polygone y, gtableau_polygone z, int nombre, float angle, float interligne) hachure le polygone.

void g3ecritblanc(gcoord x, gcoord y, gcoord z, int nbcar) efface un texte en remplissant un polygone, suivant dircar

19.6.5 Vecteurs

Cette bibliothèque, utilisée par G3D et donc de toute façon nécessaire à votre programme peut vous être utile. Elle contient la définition de la constante PI, et les fonctions suivantes :

float gscalaire(float x1, float y1, float z1, float x2, float y2, float z2) : rend le produit scalaire de deux vecteurs, **float gnorme(float x, float y, float z)** rend la norme du vecteur, **void gnormer(float *x, float *y, float *z)** modifie les coordonnées du vecteur pour le rendre de norme 1, mais toujours dans la même direction, **void gvectoriel(float x1, float y1, float z1, float x2, float y2, float z2, float *x3, float *y3, float *z3)** rend le produit vectoriel des deux premiers vecteurs, **float gangle(float x1, float y1, float z1, float x2, float y2, float z2)**; retourne l'angle entre les 2 vecteurs (en les supposant sur le même point origine), entre 0 et pi.

19.6.6 OBJets à lier à votre EXEcutable

Pour utiliser G3D, il faut inclure à votre projet : G2D.C, G3D.C, VECTEURS.C. Il faudra également, en cas d'écritures, lier G3ECR.C et GECR (GECR_STD.C ou GECR_ECR.C). Pour utiliser les polygones, on devra lier GPOLY.C et G3POLY.C.

19.6.7 EXEMPLE G3D

```
#include <stdio.h>
#include <conio.h>
#include "g2d.h"
#include "g3d.h"
```

```
#include "g3ecr.h"

void cube(void)
 {
  gcouleur(14);
  g3ligne(0,0,0,0,1,0);
  g3ligne(0,1,0,0,1,1);
  g3ligne(0,1,1,0,0,1);
  g3ligne(0,0,1,0,0,0);
  gcouleur(13);
  g3ligne(1,0,0,1,1,0);
  g3ligne(1,1,0,1,1,1);
  g3ligne(1,1,1,1,0,1);
  g3ligne(1,0,1,1,0,0);
  gcouleur(12);
  g3ligne(0,0,0,1,0,0);
  g3ligne(0,1,0,1,1,0);
  g3ligne(0,1,1,1,1,1);
  g3ligne(0,0,1,1,0,1);
  gcouleur(11);
 }

void main(void)
 {
  float xo,yo,zo,xv,yv,zv;
  char c;
  float pas;
  ginit(gtyp_ecr_par_def());
  xo=10;yo=2;zo=1;
  xv=0;yv=0;zv=0;
  pas=0.5;
  puts("A,Z,E augmenter coord X,YouZ du pt visé, Q,S,D diminuer,
idem oeil en minuscule");
  puts("ESC: fin ");
  gcouleur(8);
  gcadre();
  gcree_fenetre(0,1,0,0.92,1);
  do
   {
    gotoxy(10,2);
    printf(" XO:%6.2f YO:%6.2f ZO:%6.2f XV:%6.2f YV:%6.2f
ZV:%6.2f",xo,yo,zo,xv,yv,zv);
    g3origine(xo,yo,zo,xv,yv,zv,0,G3PERSP);
    g3echelle(−0.1,1.1,−0.1,1.1,−0.1,1.1);
    gefface();
    gcouleur(8);
    gcadre();
    gcouleur(10);
    g3axes(1.1,1.1,1.1);
    gcouleur(9);
    g3ecrit(xo,yo,zo,".Oeil");
    gcouleur(15);
    g3ecrit(xv,yv,zv,".Visé");
    cube();
pas_la_peine_de_redessiner:
    c=getch();
    switch (c)
     {
      case 'a':xo=xo+pas;break;
      case 'q':xo=xo−pas;break;
      case 'z':yo=yo+pas;break;
      case 's':yo=yo−pas;break;
      case 'e':zo=zo+pas;break;
      case 'd':zo=zo−pas;break;
      case 'A':xv=xv+pas;break;
```

```
      case 'Q':xv=xv-pas;break;
      case 'Z':yv=yv+pas;break;
      case 'S':yv=yv-pas;break;
      case 'E':zv=zv+pas;break;
      case 'D':zv=zv-pas;break;
      case 27:break;
      default: goto pas_la_peine_de_redessiner;
     }
   }
  while (c!=27);
  gfin();
 }
```

19.7 SOURIS : utilisation de la souris sous G2D

Cette bibliothèque permet d'utiliser plus facilement une souris, en utilisant G2D en mode graphique ou texte (GINIT(0)). Toutes les fonctions de cette bibliothèque commencent par S. on trouvera des fonctions de bas niveau (pour gérer directement les informations de la souris) dans l'échelle de la souris, qui commenceront par SB.

19.7.1 types utilisés

Ce paragraphe décrit les types prédéclarés qui seront utilisés pour passer les arguments aux fonctions de la bibliothèque :

```
#define SNB_BOUTONS 3
typedef int st_boutons[SNB_BOUTONS]
        ( dans l'ordre : bouton gauche,droite,centre)
/* définition des numéros de boutons */
#define SGAUCHE 0
#define SDROITE 1
#define SMILLIEU 2
```

Ces déclarations, ainsi que les fonctions décrites plus loin, seront connues de votre programme par la déclaration **#define "SOURIS.H"** et le lien avec SOURIS.C.

19.7.2 fonctions d'usage courant

Les coordonnées, si elles sont utilisées par la fonction, correspondent à vos coordonnées utilisateur, dans la fenêtre active (définie dans G2D, par Gcree_fenetre et Gechelle). On pourra assez facilement retrouver (ou recréer) ces fonctions sur tout matériel, utilisez les donc de préférence.

void sinit(int * nb_bts) : initialise la souris, retourne le nombre de boutons de la souris (0 si non installée (hard ou soft), 2 ou 3 en général). Le type de curseur est initialisé à un rectangle en mode texte, une flèche en mode graphique. Le curseur est caché, mais est positionné au milieu de l'écran.

void smontre_curs(void) : le curseur est affiché et suivra les déplacements de la souris. Il ne faut utiliser cette fonction que quand le curseur est caché.

void scache_curs(void) : cache le curseur. A utiliser par exemple pour sauver un écran graphique (sans le curseur de la souris), effacer l'écran, ou déplacer la souris par soft. Il ne faut utiliser cette fonction que quand le curseur est visible.

void sread(st_boutons b, gcoord *x, gcoord *y) : lit l'état de la souris : quels boutons sont appuyés et la position actuelle de la souris, dans l'échelle de la fenêtre graphique actuellement active, même si vous cliquez à l'extérieur de cette fenêtre (si vous n'avez pas empêché la souris de sortir de votre fenêtre). Cette fonction n'attend aucun appui de bouton, il faut pour celà utiliser un "do ... while (!b[i])" par exemple.

void sattend(st_boutons b, gcoord *x, gcoord *y) : idem SREAD mais attend que l'on appuie sur un des boutons de la souris, puis qu'on le relâche. Il rend l'état et les coordonnées mesurées lors de l'appui, mais ne retourne à la fonction appelante que lors du relâchement.

void sposit_curs(gcoord x, gcoord y) : positionne le curseur souris en x,y dans la fenêtre graphique active. On a intérêt à cacher le curseur avant, pour le remontrer après.

void sbornes(gcoord minx, gcoord maxx, gcoord miny, gcoord maxy) : limite (en X et Y) le déplacement de la souris. Il vaut mieux positionner auparavant le curseur de la souris à l'intérieur des bornes désirées. En général il vaut mieux associer une fenêtre G2D à la zonz limite et appeler :

void sbornes_fenetre(void) : limite la souris à votre fenêtre G2D actuelle.

Attention, pour ces deux dernières fonctions, le changement de fenêtre G2D ne change pas les bornes, il faut réappeler cette fonction.

19.7.3 forme du curseur

On peut modifier la forme du curseur (par défaut, un rectangle en inverse vidéo en mode texte, une flèche en mode graphique). Excepté SCROIX, ces fonctions utilisent des arguments trop proches du hard pour être facilement transportable sur un autre matériel. Ne les utilisez que si vous êtes sur de ne jamais avoir besoin de changer de matériel (ni Macintosh, ni station de travail Unix,...)

void scroix(void) : définit le curseur graphique sous forme d'une croix (au lieu de la flèche par défaut).

void scurs_graph(struct st_curseur *c) : fixe le type de curseur en graphique (voir la déclaration du type t_curseur dans SOURIS.C).

void scurs_text_ligne(int deb, int fin) : fixe le type de curseur en mode texte, sous forme d'une ligne dont on donne l'épaisseur par deb et fin (entre 0 et 7; 7,7 curseur MS DOS). Ce curseur sera déplacé par un gotoxy, contrairement au curseur graphique et au curseur texte/caractère. Il n'affecte PAS WhereX et WhereY (c'est à dire qu'un printf déplace le curseur, mais si on déplace le curseur par la souris, le prochain printf se fera quand même ligne suivante). Pour qu'un printf ne change pas la position du curseur, il faut cacher le curseur, faire le printf puis montrer le curseur, qui sera remis à l'endroit où il avait été caché. Attention, ces remarques ne sont vraies que pour le curseur text_ligne.

void scurs_text_char(int fond, int curs) : c'est le mode de curseur par défaut en mode texte. Cette fonction fixe le type de curseur en mode texte, en changeant les couleurs des caractères déjà sur l'écran. Ce curseur ne sera pas déplacé par un gotoxy et ne changera pas wherex/y. Le curseur texte/ligne reste derrière le dernier caractère écrit (ou gotoxy).

```
FOND contient :
       1 bit si clignotant
       3 bits couleur de fond (0/7)
       4 bits couleur caractère (0/15)
       8 bits code ASCII du caractère
```

seuls points restant du caractère sous le curseur : ceux appartenant à ce caractère et au caractère du fond (AND). CURS contient la même chose, mais les points restants sont ceux qui n'appartiennent qu'a soit au reste de l'application du fond, soit au caractère du curseur (XOR).

exemple : $77FF,$7700 échangent la couleur du caractère (valeur à l'initialisation).

19.7.4 fonctions de bas niveau

Les coordonnées données par la souris sont, dans la plupart des modes graphiques d'un PC, dans les plages 0−639 et 0−199. Elles sont arrondies à l'inférieur si nécessaire : en 320/200 les X sont toujours pairs, en texte 80 col on n'a que des multiples de 8. Par contre en mode 640x480 les coordonnées souris sont les mêmes que les coordonnées écran. Toutes les fonctions dans cette échelle de base commencent par SB, les Y vont de haut en bas. Il est bien évident qu'il ne faut pas utiliser ces fonctions dépendant trop du matériel (y compris du driver de souris) sauf raison particulière.

void sbread(st_boutons b, gpixel *x, gpixel *y) : lit l'état des boutons et la position : x entre 0 et 639, y entre 0 et 199, 0,0 étant le coin supérieur gauche

void sbbornes(gpixel xmin, gpixel xmax, gpixel ymin, gpixel ymax) : impose les bornes de déplacement du curseur de la souris.

void sbposit_curs(gpixel x, gpixel y) : positionne le curseur en X,Y

19.7.5 exemple

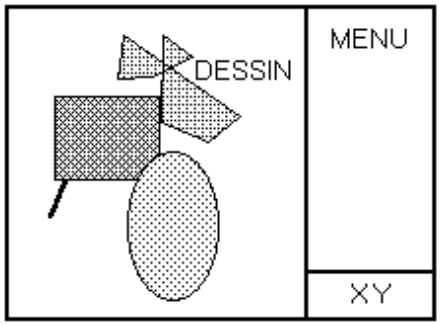
```
#include "g2d.h"
#include "souris.h"
#include <stdio.h>
#include <conio.h>

void main(void)
 {
  st_boutons b;
  int a,i,nb;
  float x,y,m,n;

  ginit(gtyp_ecr_par_def());
  gechelle(0,1,0,1);
  sinit(&nb);
  sbornes_fenetre();
  scroix();
  gotoxy(2,2);printf("vous possedez %d boutons (mais pas sur la
figure)\n",nb);
  gotoxy(1,21); /* gotoxy n'est pas une fonction standard */
  puts(" bouton gauche : repositionner curseur");
  puts("        droit  : définir une fenêtre");
  puts(" fin par CTRL BREAK ou bouton milieu");
  smontre_curs();
  gcadre();
  gcree_fenetre(0,1,0,1,1);
  gechelle(−100,100,−100,100);
  do
   {
    sread(b,&x,&y);
    gotoxy(60,1);
    printf("x:%7.2f y:%7.2f\n",x,y);
    if (b[SGAUCHE])
     {
      clreol();
      printf("nouvelle position curseur (au clavier) : x? ");
      scanf("%f",&x);
      printf(" y? ");
      scanf("%f",&y);
      scache_curs();
      sposit_curs(x,y);
      smontre_curs();
     }
```

```
    else if (b[SDROITE])
     {
      clreol();
      gchoix_fenetre(0);
      sbornes_fenetre();
      printf("cliquez votre coin inf gauche ?");
      sattend(b,&x,&y);
      clreol();printf("coin sup droit ?");
      sattend(b,&m,&n);
      scache_curs();
      gcree_fenetre(x,m,y,n,1);
      gechelle(−100,100,−100,100);
      gcadre();
      sbornes_fenetre();
      smontre_curs();
     }
   }
  while (!b[SMILLIEU]);
  gfin();
 }
```

19.8 Bibliothèque MENUG

19.8.1 Descriptif général

Cet utilitaire regroupe des utilitaires permettant de gérer toute application graphique interactive en C (ou Pascal). Il décompose l'écran en deux fenêtres : l'une graphique, de forme carrée, utilisant la partie gauche de l'écran, la partie de droite étant réservée au texte. Cette bibliothèque utilise les fonctions de G2D, GECR,...



La fenêtre graphique utilise 71% de la largeur de l'écran, ce qui fait un carré sur un écran classique. La fenêtre de menus fait 23 lignes de 21 caractères, son "scrolling" est indépendant des autres fenêtres

Toutes les fonctions graphiques seront limitées à la fenêtre graphique. Tous les écritures se feront automatiquement dans la fenêtre texte, arrivé en 23è ligne seule la fenêtre texte sera remontée (scrollée) d'une ligne.

Une gestion de menus (demander à l'utilisateur un choix entre plusieurs options) est prévue. L'option peut être choisie :

− en donnant la 1ère lettre,
− en sélectionnant une option à l'aide des flèches puis RETURN,
− en sélectionnant la dernière option par ESCAPE,
− en pointant une option à l'aide de la souris.

Le bouton gauche de la souris correspondra à un RETURN, le droit à ESCAPE.

De plus la fonction **read_coord** permet de lire des coordonnées :
− en entrant la valeur X et Y au clavier,
− en déplaçant le curseur graphique, dans la fenêtre graphique, à l'aide des flèches, puis validation par

RETURN, ESCAPE ou une touche de fonction (CTRL PGUP et PGDN permettent alors de modifier le pas de déplacement),
− en pointant le point dans la fenêtre graphique à l'aide de la souris.

la touche INS (ou bouton milieu de souris) permet d'afficher un réticule.

19.8.2 Description détaillée

19.8.2.1 Détails pratiques

Il est impératif d'inclure au sommet de votre fichier, **#include "G2D.H" #include "MENUG.H"** (plus les autres utilitaires si nécessaire). Mais il faudra lier G2D, MENUG, GECR, GPLUS

Pour initialiser le mode MENUG, appelez **void minit(gtypecr typ, gcoord minx, gcoord maxx, gcoord miny, gcoord maxy)**, qui efface l'écran et prépare la fenêtre graphique ainsi que la fenêtre de texte. Les variables réelles minx, maxx, miny, maxy doivent correspondre aux limites des coordonnées du dessin que vous désirez afficher. Typ correspond au type d'écran désiré (par exemple **gtyp_ecr_par_def()**, voir G2D).

Les textes seront écrits dans la fenêtre texte par **mprint**. Les **mgotoxy** considèrent la colonne 0 comme étant la première de la fenêtre texte (21 colonnes utilisables). Pour changer l'échelle, il faut changer les bornes MINX, MAXX, MINY, MAXY et appeler **mregen**

Si l'on désire écrire à l'extérieur de la fenêtre texte, on peut utiliser GECRIT, ou alors on peut définir la fenêtre graphique comme une fenêtre texte par la fonction **fen_text_gauche()**. Ne pas oublier de revenir à une fenêtre texte normale à droite.

La fenêtre graphique de gauche, à votre échelle, porte le ndeg. 1 dans G2D. la fenêtre texte est la ndeg. 2, l'affichage des coordonnées la 3, la 4 étant limitée à la fenêtre gauche, mais à une échelle permettant un affichage 59 caractères sur 25 lignes. Par défaut, **menug** se place et reste dans la fenêtre 2.

On quittera le mode menus par **mfin()**.

19.8.2.2 Menus

La fonction **int menu(tmenu tab, int ent, int nb, int defaut)** permet de gérer les menus. **tab** est du type prédéfini **tmenu** (tableau de 23 lignes de 21 caractères). Il contient les commentaires à afficher: lignes d'entête, options de choix. Ent est le nombre de lignes d'entête, Nb est le nombre d'options, Defaut est le ndeg. d'option proposée par défaut. La fonction rend le ndeg. de l'option choisie (entre 0 et NB−1)

Rq: on a toujours intérêt à mettre en dernier les options de type FIN, QUITTER, etc car la touche ESCAPE envoie toujours sur la dernière option.

Menu efface toujours l'écran texte lors de son appel, mais pas en sortie.

19.8.2.3 lectures (clavier ou souris)

void readst(char *nom) est une fonction permettant de lire une chaîne. Il ne pose pas de question (à vous de faire le mprint). Mais il accepte le CTRL C. De plus le RETURN peut être fait par un bouton de la souris. Un RETURN seul rend une chaîne vide (strlen(nom)=0).

void readi(int *i) lit un entier. Il repose la question si on entre autre chose. Il accepte le CRTL C et la souris. Un RETURN laisse I à la même valeur qu'à l'appel de la fonction. **void readr(float *r)** fait de même sur un réel.

La fonction **void read_coord(gcoord *x, gcoord *y, int *fonc_fin, int nb_valeurs)** permet de lire X et Y. La valeur X et Y à l'appel détermine la valeur par défaut, le curseur s'y place dans la fenêtre graphique. On peut valider directement cette position, ou déplacer le curseur par les flèches ou la souris, ou donner les 2 valeurs directement au clavier. L'appui sur INS (2 fois) ou bouton milieu souris affiche un réticule qui permet de vérifier un alignement avec d'autres points. CTRL PgUp et PgDn augmente ou baisse le pas de déplacement par les flèches. La position du curseur est constamment indiquée en bas à droite (lignes 24 et 25, sous les menus, dans la fenêtre ndeg. 3). La variable de sortie FIN indique comment la position a été validée:

FIN = 1..10 si fin par touche F1..F10
= 1 si CR ou bouton gauche souris
= 2 si ESC ou bouton droit souris
=−1 si on a donné 2 valeurs au clavier

Dernier utilitaire proposé, la fonction ERR qui affiche le message "demande rejetée", et attend CR (ou bouton souris) pour continuer, CRTL C pour arrêter

19.8.3 référence

19.8.3.1 Constante prédéfinie

nbcolmenu=21 : nombre de caractères d'une ligne de menu

19.8.3.2 Variable globale prédéfinie

int nbligmenu: nombre de lignes de menu affichables (23 normalement)

19.8.3.3 Types prédéfinis

char tstring[255] : chaîne par defaut
char tstring21[nbcolmenu] : chaîne pour les menus, nbcolmenu=21 actuellement
tstring21 tmenu[nbligmenu] : tableau comportant les lignes de menu à gérer

Les fonctions avec des arguments de type tstring ou tstring21 peuvent être appelées en donnant n'importe quel **char*** ou **char[]**. De même, si un **tmenu** est demandé, tout tableau ou pointeur de tstring21 sera accepté.

19.8.3.4 Gestion du mode MENU

void minit(gtypecr typ, gcoord minx, gcoord maxx, gcoord miny, gcoord maxy) : initialisation mode MENU, passage en mode graphique, effacement de l'écran et tracé des cadres. Ne pas réappeler cette fonction avant d'avoir fait un **mfin()**. On se trouve par défaut dans la fenêtre ndeg. 2, curseur de souris caché.

void mregen(gcoord minx, gcoord maxx, gcoord miny, gcoord maxy) : réinitialisation du mode MENU / effacer fenêtre / sur écran multifenêtre (station Unix) relecture des limites de fenêtre et recalcul des échelles.

void mfin(void) : fin du mode menu et du mode graphique.

19.8.3.5 Fonctions principales

int menu(tmenu tab, int ent, int nb, int defaut) : retourne le numéro d'option choisie. On donne le schéma d'affichage tab, le nombre de lignes d'entête ENT, le nombre d'options possibles NB, l'option par défaut DEFAUT. La fonction efface la fenêtre menu au début et en sortie. On valide par CR, on choisit par les flèches Haut, Bas, Home,End ou la première lettre. Si une souris est présente, on teste la souris ET les touches.

void read_coord(gcoord *x, gcoord *y, int *fonc_fin, int nb_valeurs) : affiche et déplace un réticule en X,Y par les flèches, fin par CR (fonc_fin=1) ou ESC (2). Si on tape une valeur, elle est prise en compte, fonc_fin=−1. Si nb_valeurs=1 : la valeur est dans X, nb=2 : dans X et Y. CTRL PgUp et PgDn augmentent ou baissent le pas de déplacement. INS affiche un réticule plein écran, jusqu'au prochain ins. Marche également avec la souris (bouton gauche=CR, droit=ESC, milieu=INS).

void affcoord(gcoord x, gcoord y, int fen_de_depart) : affiche la coordonnée dans la fenêtre de coord (ndeg. 3) puis retourne dans la fenêtre fen_de_depart. Cette fonction est appelée par read_coord, vous n'avez donc pas besoin de vous en servir

19.8.3.6 Ecritures dans la fenêtre menu

void mgotoxy(int x, int y) :uniquement dans la fenêtre menu, déplace le curseur (l'endroit où débutera la prochaine écriture). x entre 0 et nbcolmenu−1, y entre 0 (en haut) et nbligmenu−1.

void mscroll(void) : scroller la fenêtre menu de quelques lignes vers le haut. Les fonctions d'écriture appelant si nécessaire cette fonction, vous n'avez normalement pas besoin d'utiliser cette fonction.

void mprintc(char c) : affiche UN caractère dans la fenêtre menu, en position actuelle du curseur. Si l'on se trouve en fin de ligne, passe à la ligne suivante, si l'on se trouve en dernière ligne "scrolle" les textes vers le haut.

void mprint(tstring21 s) : écrit la chaîne de caractères la fenêtre menu, le curseur reste derrière le dernier caractère écrit.

void mprintln(tstring21 s) : écrit la chaîne de caractères la fenêtre menu, le curseur passe au début de la ligne suivante.

void mclrscr(void) : efface uniquement le texte (fenêtre menu)

19.8.3.7 Lectures (clavier ou souris)

void readst(char * nom) : lecture clavier d'une chaîne de caractères, rend nom initial par défaut, c'est à dire en cas d'appui de CR ou bouton gauche de la souris.

void readr(float *r) : lecture d'un réel avec test d'erreur, valeur initiale par défaut

void readi(int *i) : lecture d'un entier avec test erreur.

Les deux fonctions suivantes, utilisées par ces fonctions de lecture, peuvent également vous servir :

void err(tstring21 mess) : affiche le message d'erreur puis attend un appui de touche ou de bouton de souris.

void err_read(void) : demande si continuer ou abandonner

19.8.3.8 Ecrire dans la fenetre de gauche (58 caractères, 25 lignes)

void fen_gauche(void) : passer à la fenêtre gauche, mais à l'échelle 1..58, 1..25 (fenêtre ndeg. 4 G2D)

void efface_fen_gauche(void) : passer à la fenêtre gauche et l'effacer

void fen_droite(void) : revient à la fenêtre de menu (obligatoire avant tout appel de fonction pour le menu). ce n'est qu'un gchoix_fenêtre(2), il peut aussi être utilisé après un dessin dans la fenêtre gauche.

void mggotoxy(int x, int y), **void mgprint(tstring21 txt)** et **void mgprintln(tstring21 txt)** permettent les

écritures dans la fenêtre de gauche, à condition d'avoir fait auparavant un **fen_gauche().**

19.8.4 Exemple MENUG

```
#include <stdio.h>
#include <conio.h>
#include <string.h>

#include "g2d.h"
#include "menug.h"

void main(void)
 {
  tstring21 m[]={"entête ligne 1",
                 "",
                 "faites votre choix",
                 "",
                 "Ecrire à droite",
                 "Ecrire à gauche",
                 "Dessin",
                 "Fin du programme"};
  int i,j;
  tstring s;
  float x0,y0,x1,y1;

  for (i=1;i<100;i++) printf("écran encore normal  ");
  printf("\n\nappuyez une touche pour continuer");
  getch();
  minit(4,0,10,0,10);
/* si l'on n'avait pas initialisé m, on aurait pu le charger ainsi
:
  strcpy(m[0],"entête ligne 1");
  *m[1]=0;
  strcpy(m[2],"faites votre choix");
  *m[3]=0;
  strcpy(m[4],"Ecrire à droite");
  strcpy(m[5],"Ecrire à gauche");
  strcpy(m[6],"Dessin");
  strcpy(m[7],"Fin du programme");
*/
  do
   {
    j=menu(m,4,4,2);  /* position 2= le 3ème */
    switch (j)
     {
      case 0: mclrscr();
              for (i=1;i<30;i++)
               {
                mprint("txt droite ");
                sprintf(s,"%d",i);
                mprintln(s);
               }
              break;
      case 1: efface_fen_gauche();
              for (i=1;i<100;i++) mgprint("texte à gauche
");
              fen_droite();
              break;
      case 2: mclrscr();
              mprintln("choisissez un point");
              read_coord(&x1,&y1,&i,2);
              mprintln("choisissez d'autres");
              mprintln("points jusqu'à finir");
              mprintln("par ESC ou bouton ");
```

```
              mprintln("droit de la souris");
              do
               {
                x0=x1;y0=y1;
                read_coord(&x1,&y1,&i,2);
                gchoix_fenetre(1);
                gligne(x0,y0,x1,y1);
                gchoix_fenetre(2);
               }
              while (i!=2);
              break;
     }
   }
  while (j!=3);
  mfin();
 }
```